REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE May 2002	3. REPORT TYPE AND DATES COVERED GFLOPS Final Report (4-1-99 to 3-31-02)

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
A Flight Software Development and Simulation Framework for Advanced Space Systems	F49620-99-1-0217

6. AUTHOR(S)
David W. Miller John P. Enright

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Professor David W. Miller Massachusetts Institute of Technology 77 Massachusetts Ave., 37-327 Cambridge, MA 02139-4307	MIT SSL #5-02

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING / MONITORING AGENCY REPORT NUMBER
Dr. Robert L. Herklotz Program Manager: Software and Systems Air Force Office of Scientific Research 4015 Wilson Blvd., Room 713 Arlington, VA 22203-1954	

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Unclassified, Unre**DISTRIBUTION STATEMENT A**
Approved for Public Release
Distribution Unlimited

20040311 056

13. ABSTRACT (Maximum 200 Words)

Distributed terrestrial computer systems employ middleware software to provide communications abstractions and reduce software interface complexity. Embedded applications are adopting the same approaches, but must make provisions to ensure that hard real-time temporal performance can be maintained. This thesis presents the development and validation of a middleware system tailored to spacecraft flight software development. Our middleware runs on the Generalized Flight Operations Processing Simulator (GFLOPS) and is called the GFLOPS Rapid Real-Time Development Environment (GRRDE). GRRDE provides publish-subscribe communication services between software components. These services help to reduce the complexity of managing software interfaces. The hard real-time performance of these services has been verified with General Timed Automata modelling and extensive run-time testing. Several example applications illustrate the use of GRRDE to support advanced flight software development. Two technology-focused studies examine automatic code generation and autonomous fault protection within the GRRDE framework. A complex simulation of the

14. SUBJECT TERMS			15. NUMBER OF PAGES 308
Simulation, Satellite formation flight			16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	

Remainder of Abstract is on the back page.

TechSat 21 distributed space-based radar mission highlights the utility of the approach for large-scale applications.

A Real-Time Simulator for the SPHERES Formation Flying Satellites Testbed

Andrew D.B. Radcliffe, Raymond J. Sedwick

May 2002

SSL #7-02

A Real-Time Simulator for the SPHERES Formation Flying Satellites Testbed

by

ANDREW D.B. RADCLIFFE

ABSTRACT

A software simulator for the SPHERES formation flight testbed, the GFLOPS SPHERES Simulator (GSS), has been developed. The Synchronized Position, Hold, Engage, and Reorient Experimental Satellites (SPHERES) testbed consists of three miniature spacecraft (or SPHERES), each with their own power, avionics, navigation, communications, and propulsion. These spacecraft will operate inside the International Space Station to test formation flying, autonomy, and autonomous rendezvous and docking algorithms. The GSS runs on the Generalized FLight Operations Processing Simulator (GFLOPS), a real-time embedded hardware testbed for the simulation of distributed space systems. SPHERES flight code can be run in the simulator to test the performance of guest investigator algorithms. The simulator models the characteristics of SPHERES hardware, including thrusters and metrology sensors, and simulates the dynamics of the spacecraft. Features include the ability to simulate SPHERE-SPHERE and SPHERE-wall collisions, as well as docking between SPHERES. A 3-D viewer allows users to monitor the motion of SPHERES within the test space and log the results for later playback. A command window allows users to view telemetry from the units and send them commands. Methods of measuring flight code processor utilization are discussed. Results are presented from sample simulations that demonstrate the capabilities of the simulator. Simulations include a leader-follower control architecture, a SPHERE-SPHERE collision, passive docking, and cooperative docking. Suggestions are given for future improvements to the simulator.

Thesis Supervisor:
Dr. Raymond J. Sedwick
Dept. of Aeronautics and Astronautics

ACKNOWLEDGMENTS

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1 Motivation

A new class of satellite system architecture is being envisioned and designed that is fundamentally different from all previous ones. It involves clusters of co-orbiting satellites that work together to attain their desired objectives. This type of mission architecture is known as a distributed satellite system (DSS). Often in a distributed satellite system, the spacecraft must maintain precise relative positions with respect to each other in order to achieve the desired mission performance. Furthermore, it is easy to envision scenarios where they would have to perform a maneuver to reconfigure the shape or size of the cluster to respond to changing conditions or objectives. The acts of maintaining and reconfiguring constellation size or shape are collectively referred to as formation flying.

This is in stark contrast to the way that satellites operate today, where one large, monolithic satellite usually fulfills the entire mission. Even if the satellite is part of a larger constellation, such as a communications constellation, it is not orbiting in close proximity (ie. on the order of a kilometer) with these other satellites, nor is it trying to maintain a precise relative position with these other spacecraft.

Formation flying, and a closely related area, autonomous rendezvous and docking between satellites, bring with them many inherent challenges and risks. This is true when any untested and unproven technology is applied to space systems. The situation is exac-

erbated for formation flying and docking, because the control algorithms being developed for DSS, and the collaboration between spacecraft that will be necessary, are far more complex than anything that has been done in these areas in the past. Moreover, the consequences of failure for these technologies are great, since failure could result in collsions between multi-million dollar satellites that could render them useless.

It is easy to see that a means of mitigating the risks associated with DSS, by allowing the required technologies to be tested prior to deployment, would be invaluable. The Synchronized Position, Hold, Engage, and Reorient Experimental Satellites (SPHERES) testbed, in development at the MIT Space Systems Laboratory (SSL) and Payload Systems Incorporated (PSI), will provide this capability [Miller, 2002; Otero, 2000]. SPHERES consists of 3 miniature satellites (or SPHERES) of about 0.2 m diameter, that will operate inside the International Space Station (ISS). With these, it will be possible to test the types of algorithms needed for formation flying, autonomy, and rendezvous and docking, in a low-risk manner.

The value of the SPHERES testbed comes from the opportunity that it provides for external guest scientists to test their own algorithms on the system. This allows experts in the fields of formation flying, or rendezvous and docking, to verify their research on an actual system. There will be 24 hours of experiment time available on ISS for the SPHERES system, spread over the course of several months. It could take the form of twelve two-hour sessions, eight three-hour sessions, or six four-hour sessions and will incorporate tests from a number of different researchers from various organizations such as Draper Laboratories and NASA Goddard Space Flight Center. Although there will be ample time between tests to analyze results and make changes to the code running on the satellites, the actual experiment time is very valuable and cannot be wasted by testing code that contains bugs.

Clearly, given the limited experiment time allotted to SPHERES on ISS, the software that will run on the satellites needs to be extensively verified before hand. The best way to

verify software is to run it in a hardware-in-the-loop simulator. However, a limiting factor with space systems is that it is impossible to duplicate long-term zero-gravity on Earth. Therefore, there will always be certain types of algorithms that cannot be fully tested in a hardware-in-the-loop simulator. A solution is to employ a software simulator that allows actual flight code to be compiled into the simulator and tested in a simulated zero-gravity environment. The GFLOPS SPHERES Simulator (GSS) provides just this capability. It can run actual SPHERES flight code, while simulating the dynamics of the SPHERES units and the characteristics of their sensors and actuators.

The GFLOPS SPHERES Simulator is the subject of this thesis. This chapter first describes the objectives that were set out for the GSS. It then provides some background on envisioned DSS missions, in order to acquaint the reader with the application area for formation flying and docking algorithms. Further background on the SPHERES testbed, including other methods for pre-flight testing of SPHERES flight software, follows. Finally, an outline of the rest of the thesis is given.

1.2 Objectives

The idea for the GSS was conceived out of the need for a simulator that could test SPHERES algorithms designed for a zero-gravity, six degrees of freedom (DOF) environment. The characteristics of thrusters and sensors had to be modeled correctly, and the dynamics of the system had to be accurately represented. In addition, because rendezvous and docking would be investigated with the SPHERES testbed, the simulator needed to be able to seamlessly handle docking between units. The GSS was not seen as a tool for the initial development of algorithms. This is more likely to be done using a tool more amenable to rapid prototyping, such as MATLAB. The simulator was geared towards ensuring that the algorithm's implementation in the flight code is correct and efficient. It was also tasked with correctly representing the timing in the flight software and in the interaction of the units with other elements of the system (such as metrology beacons). To support these goals, it was deemed necessary to be able to load actual flight code into the

simulator, with as few changes as possible. In order to investigate the resource usage of algorithms, a goal was set of being able to measure their CPU utilization, as well as their memory usage. Ways were also needed to visualize and interpret the results of simulations. State histories for the units in the simulation would be needed, and a virtual 3-D viewer that showed the motion of satellites in the test space was desired as well.

1.3 Distributed Satellite Systems

Now that distributed satellite systems have been introduced, this section will explain some of the reasons why they are considered such a promising mission architecture. Some examples of planned DSS missions will also be presented.

1.3.1 Satellite Clusters

Some missions can benefit from, or can only be accomplished by, one or several clusters of satellites. A satellite cluster consists of several satellites flying in fairly close proximity (for example, a cluster with a 1 km radius), possibly with a high level of inter-satellite synchronization and communication, with payload and processing shared among the spacecraft. These satellites are usually required to maintain precise relative positions.

Perhaps the best example of formation flying satellite systems is separated spacecraft interferometers (SSI). Interferometry is the process by which electromagnetic waves from two or more apertures are combined, or interfered, to obtain an image that has better resolution than that obtained from any of the apertures in isolation. An SSI consists of two or more imaging satellites that are separated in space, yielding the same angular resolution as would be available from one large aperture with a diameter equal to the baseline between the satellites. For resolutions that require a baseline of up to a kilometer, it is clear that this cannot be accomplished by a single spacecraft, since raising such a spacecraft into orbit would not be technically feasible. Because of the precise optical pathlengths that must be maintained in SSIs, precise position and attitude control is required.

1.3.2 Benefits of Distribution

Besides the fact that they can enable missions that would otherwise be impossible, there are several other compelling benefits to using satellite clusters to perform a mission [AFRL, 2002]. The distributed architecture makes it less vulnerable to failure. As long as the cluster is designed to avoid single-point failures, by distributing critical functions across the cluster, the mission can still survive if one or more satellites cease functioning, although the overall performance will likely decrease. This graceful degradation, and the possibility to reconfigure the cluster upon failures, greatly increases reliability. Adaptability is also improved by allowing for the possibility of launching additional satellites, perhaps with new instruments, that can interoperate with the existing spacecraft. In this way, future technical advances can be incorporated cheaply instead of having to redesign the entire system, and the cluster can be upgraded easily over time. Even with an existing cluster, we can modify the cluster geometry to obtain different performance. This is particularly beneficial for imaging missions. Furthermore, manufacturing cost is decreased because of the savings brought about by mass producing several similar satellites.

1.3.3 Future DSS missions

Several DSS missions are planned over the next decade. Some are merely in the planning stage, while others are already in development. Some of these missions will now be discussed.

Starlight

Starlight is a NASA mission being developed at the Jet Propulsion Laboratory (JPL), with a proposed launch date of July 2006 [JPLA, 2002]. Starlight will serve to develop and test several new technologies and will be a precursor to future NASA missions. It will be the first spaceborne stellar interferometer and will consist of two telescopes on two spacecraft that will be separated by a distance of 40 - 600 m. The satellites will be required to maintain their separation to within less than 1 cm and their angular bearing to within 3 arcmin. The formation flying for this mission will take the form of a master/slave scenario, where

the slave, the smaller "collector" spacecraft will adjust its position and attitude in response to the motion of the larger "combiner" spacecraft. Autonomous formation flying will be tested and specific stars will be imaged. Figure 1.1 shows the two Starlight spacecraft in
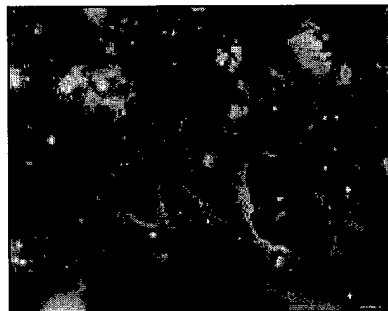
Figure 1.1 Starlight mission.

operation.

Terrestrial Planet Finder

Terrestrial Planet Finder (TPF) will take the process of imaging planets even further. It will study many characteristics of planets as far away as 50 light years, including their chemistry, in order to determine which planets have the right chemistry to support life. This will consist of measuring the relative amounts of gases such as carbon dioxide, water vapour, ozone and methane. TPF will also study how planets form from disk material around new stars. These goals will require four telescopes of 3.5 m diameter, with a baseline of 75 - 1000 m [JPLB, 2002], as depicted in Figure 1.2.

TechSat 21.

The United States Air Force is developing a distributed sparse aperture radar system for ground and air moving target indication (GMTI/AMTI) known as TechSat 21 [AFRL, 2002]. As can be seen from Figure 1.3, this system will be Earth-looking, in contrast to the systems described so far that peer into space. A flight experiment with three spacecraft is projected for 2004, while a much more extensive operational system will follow in
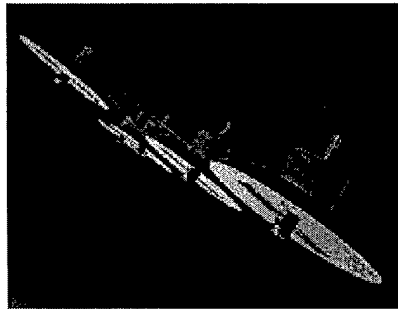
Figure 1.2 Terrestrial Planet Finder.

Figure 1.3 TechSat 21.

the future. TechSat 21 will feature extensive inter-satellite communication and complex formation flying. Because the clusters will be orbiting around the Earth, instead of residing far away from Earth like the previous systems described, it will have to counter the gravitational disturbances caused by the fact that the Earth is not a perfect sphere. Also, because TechSat 21 will be multi-mission capable, the cluster will have to be able to resize itself efficiently to adapt to different mission scenarios

Orbital Express

Another scenario that is being considered for the future is that of satellites that can rendezvous and dock autonomously. This could allow for the possibility of on-orbit refueling as satellites run out of propellant, and for hardware reconfiguration of satellites by switching

out the components of a satellite. One possibility is to have a robotic vehicle that moves between orbiting fuel depots and the satellites that it services. The Defense Advanced Research Projects Agency (DARPA) has envisioned such a system, known as an Autonomous Space Transporter and Robotic Orbiter (ASTRO) [DARPA, 2002]. Its Orbital Express Space Operations Architecture program will seek to demonstrate these capabilities with prototype technologies by 2004 and a working system by 2010.

1.4 SPHERES

1.4.1 Project Description

The SPHERES testbed will allow for cost-effective testing of algorithms for formation flying, autonomy, and rendezvous and docking. The 3 miniature satellites that make up the testbed each have their own power, on-board computer, navigation, propulsion and communications. Therefore, they will be able to move around the ISS, calculate their
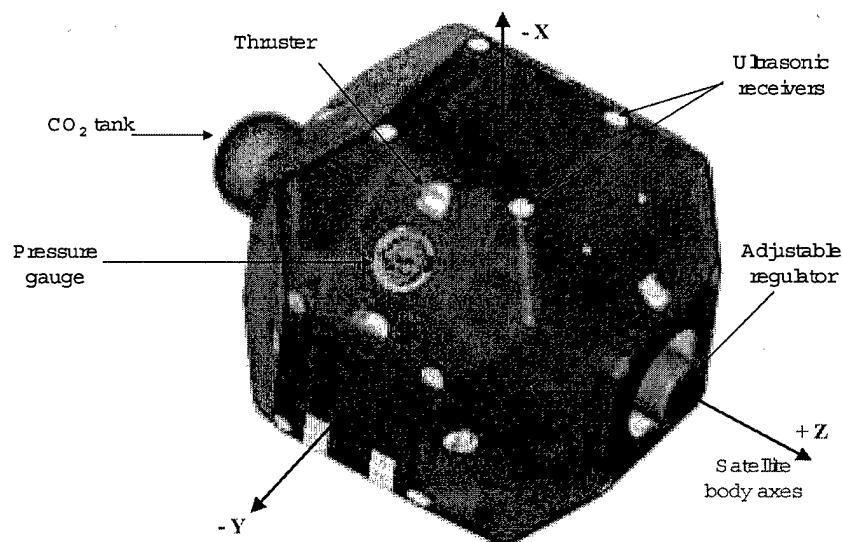
Figure 1.4 Computer-generated view of SPHERE.[1]

1. Reproduced from GSP Interface Document [HilstadA, 2002].

positions and orientations, execute control algorithms, and send commands or telemetry to each other. Figure 1.4 shows a computer generated view of a SPHERE. Other components of the system include a laptop for user interaction with the satellites, and five ultrasound transmitter beacons that are placed around the test space. The beacons allow for state determination based on the ranges between each beacon and a number of ultrasound receivers onboard the satellites. The satellites are designed to eventually operate semi-autonomously in the zero-gravity environment inside the International Space Station. The operational environment will roughly take the form of a cube with side length of six feet, although operation in a space of up to 10'x 10'x 10' is possible. Figure 1.5 depicts a typical test session. Commands are uploaded to the satellites wirelessly from a laptop, and the
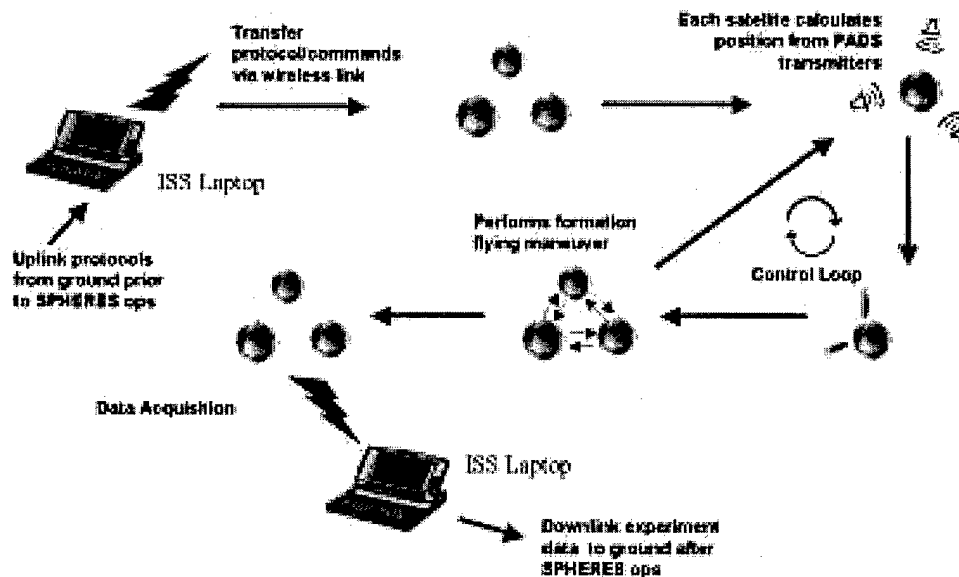
Figure 1.5 Typical Spheres Test Session[1].

1. Reproduced from SPHERES CDR presentation, Feb. 15, 2002 [Miller, 2002].

satellites, which are able to determine their own position and orientation, fire appropriate thrusters to maintain some formation flying configuration, or perform some maneuver.

All the while, the satellites communicate with each other wirelessly, and send state and debug data to the laptop. This data can then be sent back down to Earth after the test session. Interaction by astronauts will consist of starting each test by arranging the SPHERES and turning them on, sending commands from the laptop, and replenishing consumables (batteries and propellant).

The SPHERES project began as a three-term project design course for undergraduates in the Department of Aeronautics and Astronautics at MIT in 1998 and was transferred to graduate students once the course was over and the prototype hardware had been built and its operational capabilities verified. Since that time, graduate student researchers have been refining the control, communication, and metrology algorithms used in the system. Now, PSI is designing and building an improved set of hardware for the operational system to be used on ISS. SPHERES will allow high-risk algorithms to be tested, verified and improved before implementing them on complex satellites that cost tens or hundreds of millions of dollars.

1.4.2 Testing Environments

The SPHERES software and hardware needs to be extensively verified prior to its use on ISS. Besides the GSS, there are several other ways of doing this, each with its advantages and disadvantages. These verification environments will now be discussed.

Three DOF Air Table

MIT SSL has an air-table setup that allows for 3 degrees of freedom (DOF) testing of the SPHERES units. To operate on the table, the SPHERE is placed on a carriage that has three air pucks on the bottom, arranged in an equilateral triangle. Three canisters of compressed CO_2 or, alternatively, a flexible hose providing compressed air, create a layer of gas between the air pucks and the table, allowing for nearly frictionless translation and rotation. In this way, the SPHERE is allowed to translate in two directions along the 1.25 m x 1.25 m glass surface, and rotate around one axis. This is an inexpensive way to test SPHERES hardware as well as simple algorithms that do not depend on full 6 DOF

motion. A SPHERES unit can also be hung by a string above the table to verify correct state determination in three dimensions. The air table testing environment is accessible at all times by the MIT SPHERES team, allowing for iterative development, with immediate testing and verification of new code. However, with only 3 DOF, it is not possible to test more complex algorithms. There are also some disturbances present in this setup that are not expected in the ISS operational environment. Imperfections in the table surface and build-up of material on the surface cause friction, while misalignment of the surface normal with the gravity vector causes a constant drift.

Johnson Space Center Reduced Gravity Program

Another testing environment that has been used is the Reduced Gravity Program at NASA's Johnson Space Center in Houston, Texas. This program utilizes a specially modified KC-135A turbojet transport (similar to a Boeing 707) performing parabolic arcs to create 20 - 25 second weightless periods. A typical flight lasts for 2 - 3 hours and consists of 30 - 40 parabolas. Figure 1.6 shows the gravity experienced at different points in a typical parabolic maneuver. Further information can be obtained from the JSC Reduced Gravity Program website [JSC, 2002]. These flights allow for verification of the operation of hardware in micro-g, as well as investigation of the performance of algorithms in 6 DOF. However, the fact that each weightless period lasts for only 25 seconds limits the length of the tests that can be performed. Also, turbulence affects the relative motion of the SPHERES inside the frame of reference, and since the nose of the plane goes from pointing up 45° to pointing down 45° during the weightless period, the frame of reference rotates 90° during this time. These effects limit the fidelity of tests that can be performed in the KC-135. Another troublesome factor is that team members in close proximity to the SPHERE who are helping to perform tests affect the measurements used by the SPHERE to sense its position and attitude. As will be explained later, these measurements rely on ultrasound pulses that propagate to the SPHERE from transmitter beacons placed inside the test environment. Blockage of these waves by team members affects the results of experiments. This problem will be less severe on the ISS since the continuous zero-grav-
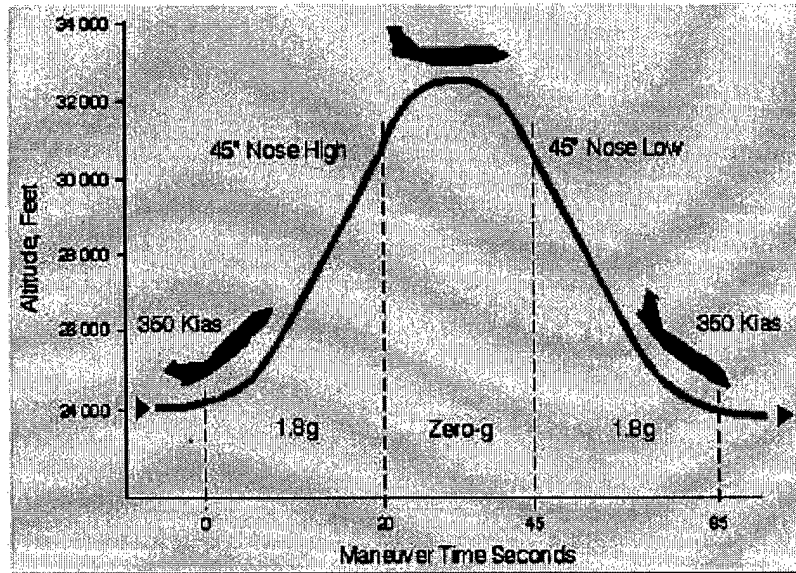
Figure 1.6 Gravity experienced during parabolic maneuver by KC-135.

ity will mean that astronauts will not have to be in close proximity to catch the SPHERES units every 25 s. But the most significant factor affecting the usefulness of the Reduced Gravity Program is that flights by the SPHERES team occur only a few times a year. Therefore, the limited availability of this test environment means that it cannot be used for regular and/or unexpected testing and verification of SPHERES software. It is primarily useful for validation of hardware.

1.4.3 Software Simulators

Since the SPHERES hardware is not likely to change after the new hardware is introduced, the above testing environments are adequate for verifying the operation of hardware. Some portions of the software are likely to remain static as well. However, many critical software functions, such as control algorithms, will be modified regularly, and will even be switched with completely new code. This is unavoidable due to the nature of the guest scientist program, which is designed to let separate investigators test independent algorithms that they design. We have seen that the above two test environments are not

adequate for validating SPHERES code. The 1g testbed can only test the 3 DOF performance of algorithms, while the KC-135 test environment is plagued by limited flights and other problems. Clearly, there is a need for the capability to test code developed for the testbed on a regular basis, in a manner that closely represents the characteristics of the ISS operating environment. Since there is no feasible way of simulating a zero-g environment with the real hardware (other than on the KC-135), then a software simulator is the only way to fully test code developed for SPHERES. A software simulator can easily simulate a zero-g environment and is highly accessible for testing. The SPHERES Guest Scientist Program (GSP) provides two software simulators for guest investigators.

GSP Simulation

The GSP simulation, developed by Mark Hilstad, is intended to aid initial code development by guest scientists. It is described in detail in the SPHERES GSP Interface [Hilstad A, 2002]. It includes the ANSI C source code files that make up the SPHERES software framework, into which the investigator's algorithms are added. The only change made to these files is that some of the low-level functions that access hardware have been modified. The GSP simulation is meant to allow investigators to check that their code compiles into the SPHERES software framework, and to verify basic operations in a low-fidelity simulation (either 1-g or 0-g). Guest investigators should be able to accomplish these objectives without interaction with the SPHERES team. The output of the GSP simulation is a DAT file that contains state histories and debug info. A MATLAB m-function is provided that can read this DAT file and plot state histories and the histories of quantitative debug variables.

GFLOPS SPHERES Simulator

The other simulator is the GFLOPS SPHERES Simulator, the subject of this thesis. The Generalized FLight Operations Processing Simulator (GFLOPS) is a testbed that consists of hardware and associated software that was designed for real-time distributed satellite system simulations. Located in the MIT Space Systems Laboratory, it consists of eight

networked PowerPC single-board computers running the OSE real-time operating system. Actual SPHERES code, with minor modifications, can be run on these computers. The GFLOPS SPHERES Simulator propagates the dynamics of the SPHERES satellites and provides metrology information to the SPHERES. It allows for simulation of motion in 6-DOF, in a 0-g (free-flying) or 1-g (air table) environment, with multiple satellites. Noise can be added to thruster firings or metrology readings in order to better approximate the actual SPHERES operational environment. Alternatively, the satellites can be provided with perfect state knowledge to investigate the performance of a formation flying algorithm under best-case conditions. The simulator can provide debug data and state histories from test runs. Furthermore, a viewer allows the results of test runs to be visualized in a 3-D environment on a PC, either as the simulation progresses, or later in a playback mode. The simulator also allows for commands to be sent to the satellites from a PC, just as in an actual operational scenario. An advantage of this simulator is that its real-time nature allows for better representation of the timing and interrupts of the actual SPHERES system than the GSP simulation.

The four components of the SPHERES Guest Scientist Program are depicted in Figure 1.7. The KC-135 tests are not considered part of the program since they occur too infrequently and are used primarily for hardware verification. The accessibility and fidelity of the various elements of the program vary inversely. The software simulators located at the top of the figure are suitable for less mature algorithms, where we want to investigate the general performance of the algorithm, to verify that nothing goes catastrophically wrong. At this stage of development, a realistic disturbance environment is not necessary and may even decrease the usefulness of tests. We desire high accessibility to the testing environment at this point, because of the iterative nature of the early stages of algorithm development. The air table and the Space Station testing environments have much more fidelity, but they are less accessible for testing. The air table requires the time of SPHERES team members, while the ISS testing is limited to 24 hours of total operations. They are not as conducive to the initial algorithm development and testing needs of guest scientists. However, for mature algorithms that can be tested in 3 DOF, the air table is a
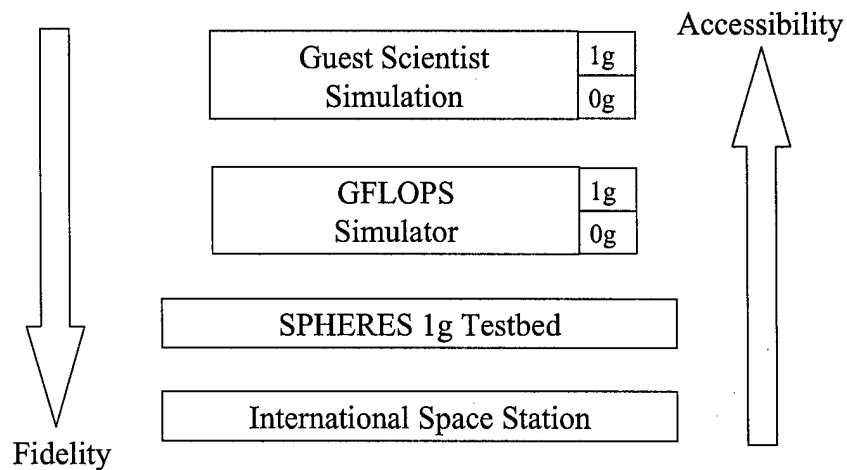
$$Utility = f\,(fidelity,\ accessibility,\ algorithm\ maturity)$$

Figure 1.7   Guest Scientist Program components.
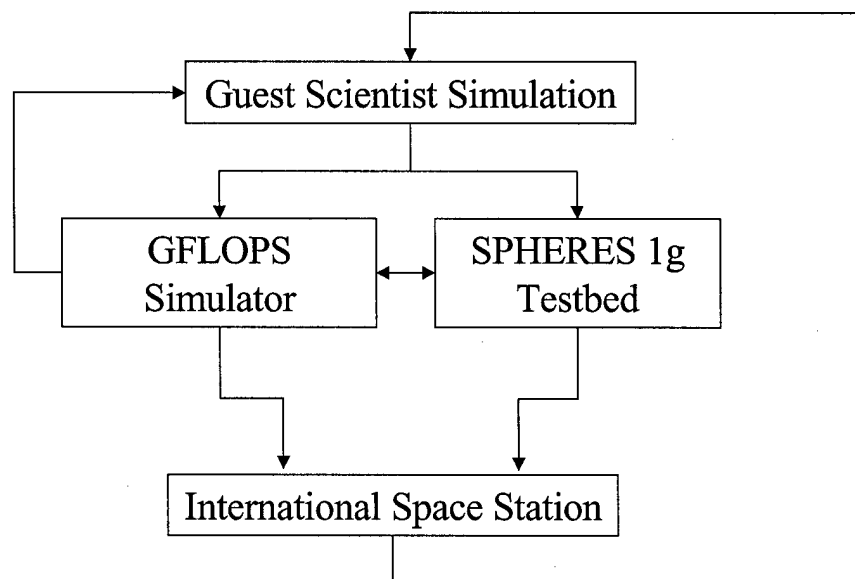
Figure 1.8   GSP testing sequence.

valuable verification environment since it allows testing with actual SPHERES units. This is essential for refining control parameters that depend on the precise characteristics of the SPHERES units. Figure 1.8 illustrates the iterative nature of testing within the GSP

program. While testing begins at the GSP Simulation for a particular algorithm, it can progress from there through any path through the various GSP elements. Data from testing on the International Space Station can even be used to refine algorithms for future sessions.

1.5 Outline of Thesis

This thesis provides a comprehensive description of the GFLOPS SPHERES simulator. Chapters 2 and 3 contain essential background information. First, Chapter 2 gives a detailed description of the SPHERES formation flying testbed. This discussion is very important because it outlines those features of the SPHERES hardware that are modeled in the simulator, as well as the software that must run in the simulator. Then, Chapter 3 describes both the hardware and software that make up GFLOPS, the real-time distributed spacecraft simulation testbed on which the simulator operates. Chapter 4 is the main section in the thesis. It describes each of the software modules that make up the simulator, as well as the overall architecture into which they fit. It yields a thorough understanding of the design and operation of each of these modules, as well as their interfaces with each other. Chapter 4 also gives an account of ways that we can investigate the use of resources, such as processing time or memory, by SPHERES flight code running on the simulator. Chapter 5 contains simulation results that have been obtained thus far. Several simulations were performed, providing insight into the uses of the simulator, and its accuracy in representing the SPHERES testbed. Finally, Chapter 6 sums up the work that has been accomplished thus far, puts it into context, and provides suggestions for future work that could further improve the effectiveness and accuracy of the GFLOPS SPHERES simulator.

Chapter 2

SPHERES

2.1 Introduction

This chapter describes the SPHERES formation flying testbed in detail. Since this is what we are trying to simulate, it is important to have a good understanding of the SPHERES hardware and software. We begin by examining the types of control architectures that can be implemented with SPHERES. Then each of the SPHERES subsystems is discussed, with emphasis on the details that are modeled or implemented in the GFLOPS SPHERES Simulator.

2.2 Testing Scenarios

There are several testing configurations that are available with the SPHERES testbed [Miller, 2002].

- **Independent Control**: With a single satellite, we can investigate long-term station-keeping, as well as minimum propellant maneuvers involving rotations, translations, or both. These same maneuvers can also be done in multi-satellite tests. Each satellite determines its control actions based solely on information about its state and its objectives.

- **Master/Slave Control**: We can have a master/slave control scheme, where the "master" satellite receives the state information from all satellites and decides on the action for each of them to take. The "slaves" simply send state data to the master and receive commands in return, performing no control law computation themselves.

- **Leader-follower control**: This involves the leader sending its state to the follower satellites. The followers attempt to track the leader's state, with some offset to avoid collisions.

- **Distributed control**: Distributed control is more complex, with each satellite having knowledge of the state of all the others, and determining its own motion based on this information.

These forms of control can be used to test a wide range of algorithms, including ones for autonomous rendezvous and docking, collision avoidance, and fuel balancing maneuvers. Retargeting and image plane filling maneuvers can also be performed.

2.3 Physical Properties

The physical properties of a SPHERES unit are not the same on the laboratory air table and in micro-gravity. The reason for this is that on the air table, the carriage on which the unit rests must be considered part of the SPHERE when measuring its physical properties. This affects the mass of the SPHERE, as well as its inertia matrix. The mass and inertia properties of a prototype SPHERE [HilstadA, 2002] are given in Table 2.1, for the 6-DOF

TABLE 2.1 Mass and inertia properties of prototype SPHERE for 6-DOF configuration.

Property	Value	Accuracy
Mass	3.4447 kg	± 0.001 kg
Inertia (body x-axis)	0.0204 kg m^2	± 5 %
Inertia (body y-axis)	0.0170 kg m^2	± 5 %
Inertia (body z-axis)	0.0190 kg m^2	± 5 %

TABLE 2.2 Mass and inertia properties of prototype SPHERE for air table configuration.

Property	Value	Accuracy
Mass	5.5299 kg	± 0.001 kg
Inertia (body z-axis)	0.0311 kg m^2	± 5 %

configuration, and in Table 2.2 for the air table configuration. Values for the flight SPHERES were not available at the time of publishing of this thesis.

2.4 Subsystems

A SPHERES unit consist of six main subsystems: power, software, communications, metrology, avionics, and propulsion. These will now be described. The SPHERES Critical Design Review [Miller, 2002] is a good source for more information about power, metrology, avionics and propulsion. More in-depth discussion about communications can be found in [Otero, 2000], while the best sources for software and metrology are [HilstadB, 2002], and the GSP Interface Document [HilstadA, 2002].

2.4.1 Power

The power subsystem consists of two AA alkaline battery packs that provide the power that drives the operation of a SPHERES unit. The power subsystem is not modeled in the GFLOPS SPHERES Simulator.

2.4.2 Software

The SPHERES flight software runs on a basic operating system with hardware support functions. There are three different frameworks within which guest scientists can organize their code. The **Standard Control Interface** is the most easy to use. It is made up of three main interrupts (propulsion, communications, and control) as well as background processing. When employing the Standard Control Interface, the guest scientist makes changes only to the control interrupt. This interrupt consists of standard code blocks with predefined inputs and outputs that must be adhered to. The programmer simply inserts custom control code within the predefined blocks, and/or calls functions previously coded by the SPHERES team. This process is designed to avoid modifications to the basic operation of the interrupt, thereby minimizing errors in timing or interaction with other interrupts.

The **Direct Control Interface** involves the replacement of the controller interrupt code and/or background processing code with custom logic. It places no real restrictions on the custom code. Therefore, it can be quite different from the Standard Interface, with simi-

larities existing only in the timing and general purpose of the interrupts. Thus, the Direct Interface provides strong flexibility to investigators. However, it requires them to attain a high level of understanding of the SPHERES software, especially with respect to the interactions between interrupts.

The **Custom Control Interface** places essentially no restrictions on the SPHERES software. The number, purpose, and timing of the interrupts can be changed, affording exceptional freedom to the investigator.

The four interrupts that make up the Standard Control Interface will now be described.
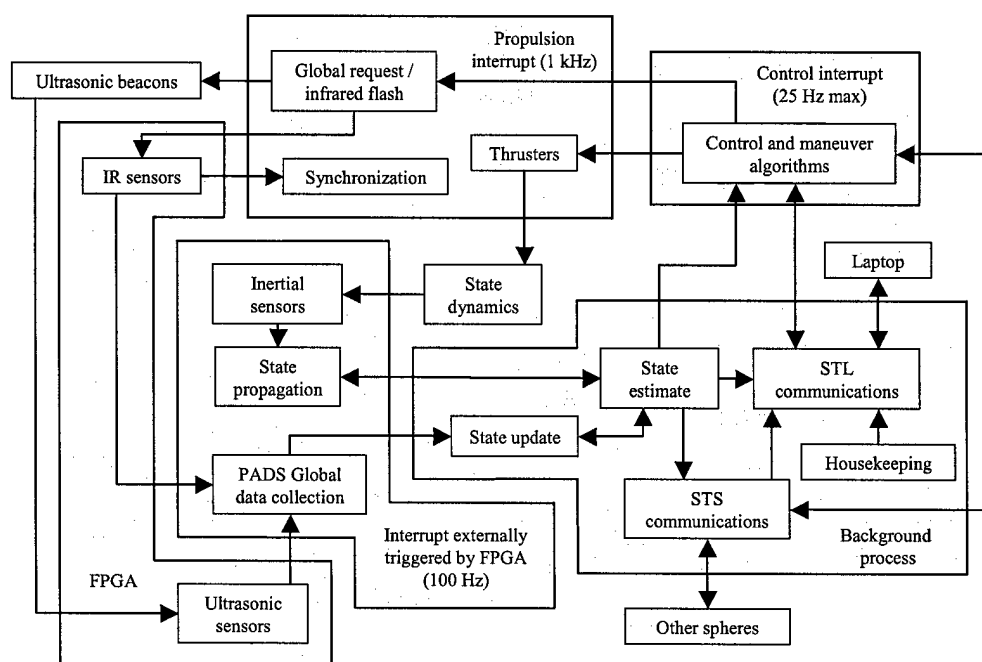
Figure 2.1   Custom Control Interface.[1]

1. Figure courtesy of Mark Hilstad.

Figure 2.1 shows the relationships between the various interrupts and functions of the Standard Control Interface.

Propulsion Interrupt

The highest priority interrupt is the propulsion interrupt (or thruster interrupt), which runs at 1 kHz. This interrupt's main objective is to determine whether to turn each thruster on or off, which is decided by checking a global array that holds the time remaining for the current pulse of each thruster. The pulse times for the thrusters are set in the controller interrupt. These times are decremented each time through the thruster interrupt, until they reach zero and the thruster is turned off. The thruster interrupt also ensures that all thrusters are turned off when the satellite is receiving global metrology, since the ultrasonic noise produced by the thrusters can impair global metrology readings. Further tasks of the thruster interrupt include incrementing time, battery and fuel usage counters, and sending requests for new global metrology information at a set period, usually 500 ms. Because of the 1kHz period of the interrupt, there is a 1 ms pulse-width resolution for the propulsion system.

Communications Interrupt

The second highest priority interrupt is the communications interrupt. This is not a timed interrupt. It executes only when communications data becomes available. It then fetches the data, checks its source, and places it in the appropriate global array according to its source (laptop, other satellite or global metrology reading), so that it can be accessed by the other interrupts.

Controller Interrupt

The controller interrupt runs at the lowest priority. It can run at any integer frequency up to 25 Hz, but usually runs at 10 Hz. The reason for the maximum frequency of 25 Hz is that heat dissipation concerns in the thrusters limit the pulse-width frequency and hence the control frequency [HilstadA, 2002]. The controller interrupt runs the desired control algorithm and determines the lengths of thruster pulses. It also processes communications received from other satellites and creates telemetry data that is ready to be sent to other satellites or ground. Intersatellite communication is important for leader/follower archi-

tectures, where the follower needs to know the state of the leader so that it can attempt to follow it.

Background Processing

Whenever none of the above interrupts are executing, background processing occurs. This consists of processing the communications data from the laptop that was placed into global arrays in the communications interrupt, sending communications (of data usually created in the controller interrupt) to other satellites or ground, performing housekeeping tasks (such as checking battery and propellant tank levels), and running the position and attitude determination routine that processes metrology information to determine the satellite's state.

2.4.3 Communications

Communications in the SPHERES testbed consists of two wireless data transmission modes: satellite-to-satellite (STS) and satellite-to-laptop (STL). These occupy different communication channels, with the STS operating at 916.5 MHz and the STL at 868.35 MHz. While these channels are bi-directional, they are half-duplex, meaning that only one unit can transmit at a time. Additionally, when a message is sent, it is received by each of the units. They must determine for themselves if the message is intended for them by looking at the destination information sent along with the message. To ensure that only one unit transmits at a time, token ring networks are used (one for STS, another for STL). These operate through the passing of a token around the network and the stipulation that a unit can only transmit when it holds the token.

Communications is in the form of packets. A packet consists of a header, the data, and a tail as depicted in Figure 2.2. Packets are divided into bytes. The *start* byte signals the start of a new packet. The *to* field specifies the intended recipient of the data and is used by units to check if a message was intended for them. The *from* field denotes the sender of the message, while *type* signifies whether it is a command, telemetry, or token message, and *size* refers to the number of bytes contained in the data section. The tail consists of

start	to	from	type	start	time1	time2	data	...	data	checksum1	checksum2

Figure 2.2 Packet configuration.

two checksum bytes, one representing the upper eight bits of the checksum, and the other the lower eight bits. The checksum will be equal to the sum of all data in the message, otherwise an error has occurred in the transmission. The packet is broken up into eight bit pieces that are transmitted separately. Commands are acknowledged in their entirety, and a good acknowledgment results in the sending of a GO command to instruct the commanded satellite to execute the command. If proper acknowledgment is not received, the command is resent. Telemetry takes the form of state and non-critical error information and is not acknowledged. That is, telemetry data that has a checksum error is simply thrown away.

2.4.4 Metrology

The metrology subsystem enables the SPHERES units to maintain knowledge of their state. The state of a SPHERES unit consists of its position and velocity with respect to the global frame, orientation within this frame (expressed via a four-element quaternion), and angular rotation rates about its three body axes. There are two metrology systems that enable determination of this state. While they each could theoretically operate by themselves, they are used in combination.

The inertial measurement unit (IMU) consists of a 3-axis accelerometer and three single-axis rate gyroscopes. All of these are aligned with the body axes defined for the SPHERE. By integrating the output of the accelerometer, the velocity and position of the unit can be recovered, while the rate gyro can yield the angular velocity and orientation. However, due to the double integration, the accelerometer, by itself, can maintain reasonably accurate position information for only about two seconds, while the rate gyros are useful for approximately 20 minutes of independent operation [Otero, 2000]. The performance

specifications for the Honeywell Q-Flex accelerometer are given in Table 2.3, while those for the BEI Gyrochip II rate gyros are given in Table 2.4 [Miller, 2002].

TABLE 2.3 Honeywell Q-Flex accelerometer performance specifications.

Input Range	±20 g
Bias	<20 mg
Scale factor	2.75 mA/g ±1.8%
Threshold and resolution	<5 μg
Bandwidth	<200 Hz
Noise 0 to 10 Hz 10 to 500 Hz	 20 μg RMS 200 μg RMS
RSS bias and scale factor one-year repeatability	1 mg
Operating temperature	-40 to 160°C

TABLE 2.4 BEI Gyrochip II performance specifications.

Input range	±50°/s
Full range output (nominal)	0 to +5 VDC
Scale factor, scale factor calibration	30 mV/(°/s), ±2 % of value
Scale factor over temperature (dev. from 22°C)	<0.06%/°C
Bias calibration (at 22°C)	+2.5 ±0.045 VDC
Short term bias stability (100 s)	<0.05 °/s
Bandwidth	>50 Hz
Output noise (DC to 100 Hz)	<0.05 °/s/(Hz)$^{1/2}$
Operating temperature	-40°C to +85°C

Because the inertial measurements lose accuracy over time, an independent method, known as global metrology, is used for first-order corrections to the state. The global metrology system works in a manner similar to the Global Positioning System. Five beacons that transmit ultrasound pulses are placed around the test space. Each SPHERES unit

has four ultrasound receivers mounted on each of six of its faces. By measuring the time of flight for ultrasound pulses to travel from the beacons to the receivers (and hence the range), the SPHERE is able to determine its position and orientation. The sequence goes as follows. At a set frequency between 0 and 5 Hz (usually 2 Hz), a "master" satellite requests a global metrology reading by sending out a pulse from an infrared transmitter. The beacons detect this pulse and send out their ultrasound pulses in a predefined sequence. First there is a 5 millisecond delay, then the five beacons transmit pulses one at a time, with 20 millisecond delays in between. Since the SPHERES also have infrared receivers that detect the initial infrared pulse by the master, and they know the timing of the ultrasound pulse transmissions, they are able to determine the time of flight of the pulses, and hence the range between receiver-beacon pairs. They are then able to compute position and attitude from this data, since they know the positions of the beacons. Since there are uncertainties in the global measurements, they are combined, using Kalman filtering, with the state that is estimated from the inertial measurements, to yield the new state. For a detailed explanation of the algorithms used to determine position and attitude from global metrology, see [HilstadB, 2002].

The intensity of the ultrasonic transmitters and the sensitivity of the receivers depend on angle. As we move away from the direction in which the transmitter or receiver is pointing, the intensity or sensitivity decreases, as shown in Figure 2.3 for the transmitter, and Figure 2.4 for the receiver. Each ultrasound emission is actually a series of pulses, with the initial ones being of lower intensities. Since the SPHERES ultrasound detection circuitry uses threshold-based detection, off-angle measurements yield time-of-flight values that are too large. To negate this effect, the flight code applies a correction based on calibrations of the transmitters and receivers. The intensity of received pulses also decreases with distance between the transmitter and receiver, so the correction takes this into account as well. After the correction, individual distance measurements from transmitter to receiver are believed to be accurate to ±3 mm [HilstadA, 2002].
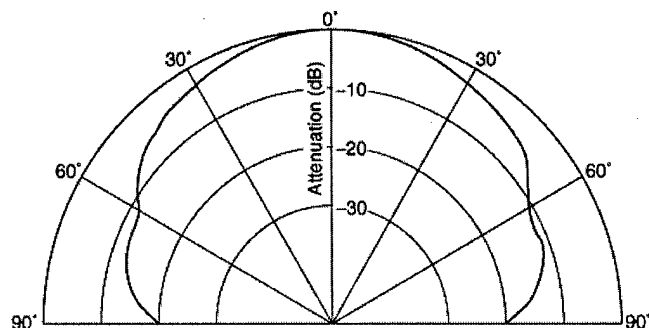
Figure 2.3 Ultrasonic transmitter intensity angle dependence.
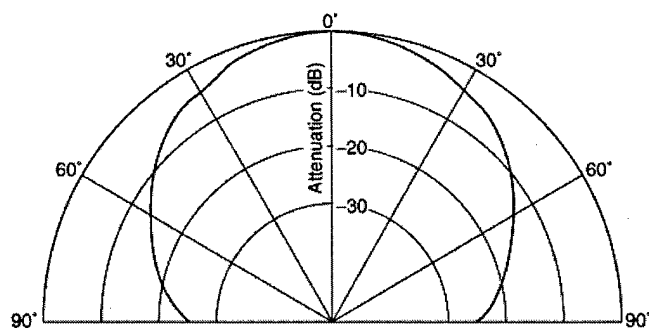
Figure 2.4 Ultrasonic receiver sensitivity angle dependence.

2.4.5 Avionics

A schematic of the SPHERES avionics is provided in Figure 2.5. A Sundance SMT375 board, featuring a Texas Instruments TMS320C6701 digital signal processor, runs the SPHERES flight code, including all guest investigator algorithms. It interfaces with a motherboard that provides electronics for communications, interfacing with sensors, and digital I/O. The DSP communicates with the motherboard by way of TIM40 standard communications ports and a global bus. The motherboard has a Xilinx Spartan II field-programmable gate array (FPGA) that provides support for local sensors (accelerometers
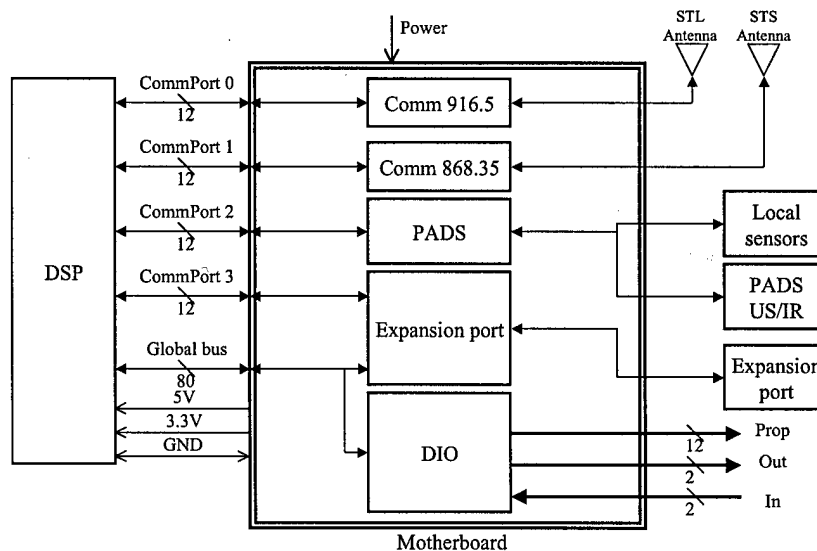
Figure 2.5 SPHERES avionics.[1]

1. Reproduced from SPHERES CDR presentation, Feb. 15, 2002 [Miller, 2002].

and rate gyros) and global metrology. The FPGA has 12 infrared receiver and 24 ultra-sound receiver input channels, as well as six 12-bit A/D channels. The FPGA calculates the distances from ultrasound beacons to receivers based on the times that inputs are received. The digital I/O on the FPGA consists of twelve outputs to control the propulsion solenoid valves, two general outputs (used to enable an LED and reset a watchdog timer), as well as two general inputs (used to detect low battery signals and to detect external ports).

2.4.6 Propulsion

The SPHERE controls its state by firing its thrusters. The thrust comes from expelling CO_2 gas, supplied from a liquid propellant tank, through machined nozzles. The thrusters are actuated by electric solenoids that control micro-valves. They operate in a binary fashion: they are either on or off. There are non-linear transients at the start and end of a pulse, when the thrust is ramping up to or down from its nominal value. A pressure regulator ensures that the pressure of the gas reaching the nozzles is some constant value that can be

set between 0 to 50 psig at the start of a test, providing for stable thrust throughout a test. A single CO_2 tank lasts for approximately 10 minutes of normal operations. Since the thrusters can only operate at one nominal force level (0.2 N at 50 psig), they are pulse-width modulated to achieve a greater range of average forces. When a thruster that is off is commanded on, the electric solenoids must have voltage applied to them for approximately 6 ms before the valve will open. No thrust is observed during this time. Therefore, the minimum commanded pulse width is set at 5 ms in the software. Once a valve begins to open, it takes approximately 1.1 ms for the thrust to reach its nominal value. Since the thruster interrupt runs at 1 KHz, the pulse width resolution is 1 ms (ie. we can choose when to turn off the thrust to within 1 ms).

2.5 Summary

This chapter provided an overview of the SPHERES testbed, with particular attention paid to those areas that are of most importance to the GFLOPS SPHERES Simulator. First, the basic control architectures that can be tested with SPHERES were outlined. Then, the main features of each of the subsystems of a SPHERES unit were described. This information is essential for understanding Chapter 4, which will discuss how the subsystems were modeled and/or implemented in the GFLOPS SPHERES Simulator.

Chapter 3

GFLOPS

3.1 Introduction

This chapter introduces the Generalized FLight Operations Processing Simulator (GFLOPS). It begins by describing the hardware for this real-time testbed, then moves on to the software. The software includes both the real-time operating system that is used, as well as real-time middleware that was designed to facilitate the development of spacecraft flight software and simulations. This real-time middleware is known as the GFLOPS Rapid Real-Time Development Environment (GRRDE).

The Generalized FLight Operations Processing Simulator (GFLOPS) is a testbed for the simulation of distributed systems, especially space systems. GFLOPS was originally developed as the doctoral thesis work of Enright [Enright, 2002], and is now available for more general use in the MIT Space Systems Laboratory. The usefulness of the GFLOPS testbed has been well demonstrated with previous studies. The most extensive project captured the behavior of the United States Air Force's TechSat 21 distributed aperture radar satellite system. The spacecraft software in this simulation included orbit control, attitude control, and radar processing, while the simulator software included dynamics, radar, sensor, and actuator simulation. This project validated the hardware and software that make up the GFLOPS testbed.

3.2 Hardware

The GFLOPS hardware consists of eight PowerCore-6750 single-board computers manufactured by Force Computers. The IBM PowerPC 750 processors on these boards run at 400 MHz and have access to 256 MB of RAM each. They are interconnected by 100 MBps Ethernet, with a 3Com SuperStack 100Base T switch. There are up to 3 support personal computers (PCs) that can communicate with this embedded hardware. This allows for simulation monitoring from PCs, sending of commands to the embedded hardware, and loading and debugging of programs. The hardware architecture is depicted in Figure 3.1.
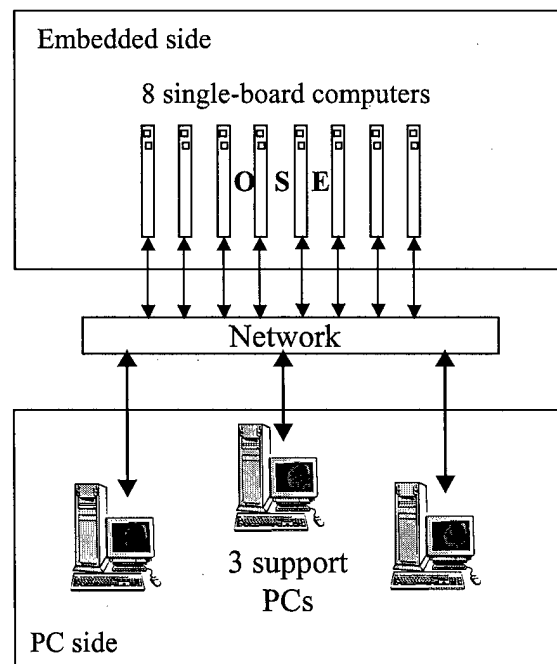
Figure 3.1   GFLOPS Hardware.

3.3 OSE Real-Time Operating System

A real-time system is one that must complete activities and respond to external events while meeting timing requirements. A hard real-time task is one that must be completed

before its deadline, or severe consequences can arise. For example, an automatic pilot system for an aircraft must update control actions at a minimum rate in order to maintain stability. Soft real-time tasks do not bring drastic consequences if a deadline is missed, but performance is degraded. For example, a DVD player should update the movie frame at a specified rate. If it misses a few frames, the movie quality will suffer, but there will not be any greater consequences. Many PC applications with which we are familiar are not real-time. While it is desirable to produce a result as quickly as possible, there are no ill effects if it takes a bit longer than expected. An example might be a compiler: we wish the program to be compiled quickly, but it does not really matter if it takes a few seconds longer than expected.

A real-time operating system (RTOS) is one that is designed to host real-time applications. It provides timing functions, as well as mechanisms for scheduling processes and switching between them. The GFLOPS testbed employs the OSE RTOS by ENEA OSE Systems, which is well suited to distributed real-time systems. OSE provides two types of kernels: a real-time kernel than runs directly on embedded systems, and a soft kernel that emulates the OSE operating system on a host PC. This allows a system to be distributed across embedded hardware and desktop PCs. OSE comes with a real-time interface known as Illuminator that facilitates loading and monitoring of applications.

Several different types of processes are available in OSE. It allows for both static processes, which are created only when an application is loaded and exist until it is killed, and dynamic processes, which can be created or killed by program code at any time. Static or dynamic processes will all be of one of the following types:

- **Interrupt** processes are triggered when there is a hardware interrupt or a specific software event (such as the sending of a signal to the interrupt process).
- **Timer interrupt** processes run at a preset interval.
- **Prioritized** processes run as an infinite loop that continues to execute until the process is interrupted.

- **Background** processes run whenever no other processes have control of the processor.

A block is a higher-level object for organizing programs. It consists of a number of processes and a memory pool that they use. The memory on a board can be partitioned into segments, with each segment providing the address space for the pools of one or more blocks. These associations are depicted in Figure 3.2.
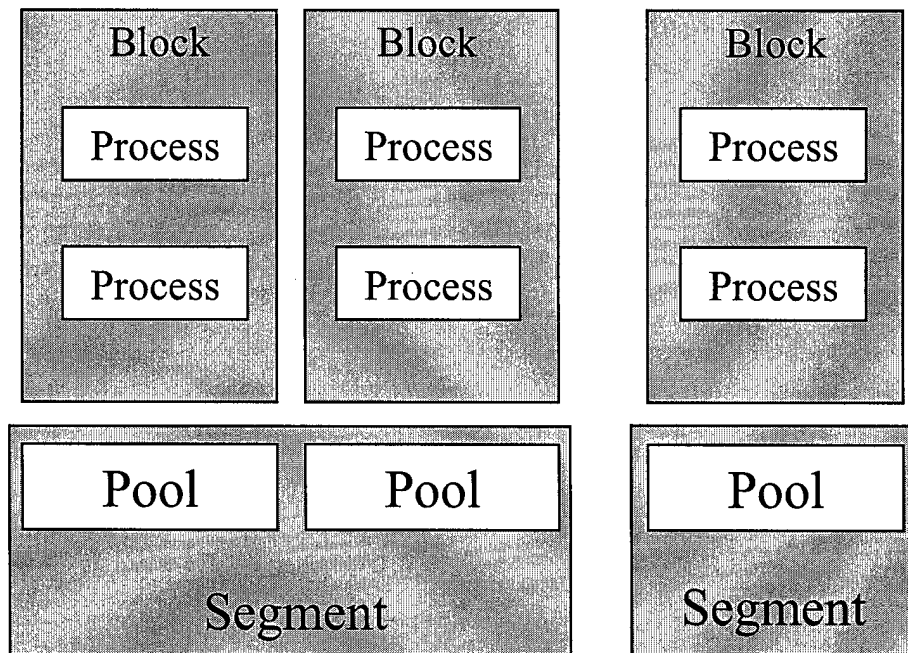
Figure 3.2 Relationship between processes, blocks, memory pools and memory segments.

In OSE, interprocess communication occurs through message passing. These interprocess messages are known as *signals*. A signal's type is specified by its signal number. In order to transmit a signal, the sending process must find the process identifier of the destination process, allocate memory for the signal, set its signal number, and fill the buffer with data. Each process has an input queue into which incoming signals are placed. The programmer can choose to selectively receive incoming messages with particular signal numbers. In addition, processes can have a redirection table that redirects some subset of incoming sig-

nals to other processes. OSE has directory services that provide the opportunity to register services, obtain their process IDs, and subscribe for notification of changes. In this way, a process that wishes to communicate with a particular type of service can find it in the directory if that process has registered itself. The scenario is depicted in Figure 3.3.
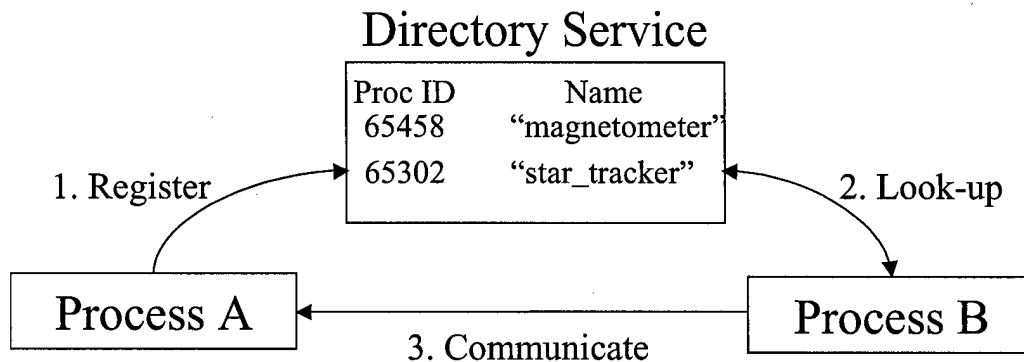
Figure 3.3 OSE Name Service.

More information regarding the OSE RTOS can be obtained from the OSE User Manuals [ENEA, 1998].

3.4 GRRDE

The GFLOPS Rapid Real-Time Development Environment (GRRDE) extends the services of the OSE real-time operating system. One of the most important functions of this real-time middleware is to enhance interprocess communication. However, it also promotes program organization and provides an object oriented interface to many OSE entities, such as processes and signals. This organization into objects simplifies coding, because we need only to search through the definition of an object's class to find all the related functions. Also, extra functions provided with the GRRDE objects make coding and debugging more efficient.

Other facilities that GRRDE provides include simulation tools, timers, synchronization objects, random number generators, and atomic objects. Atomic objects are guaranteed to be accessible for read/write operations by only one process at a time. A priority ceiling protocol is used to avoid priority inversion when using atomic objects [Enright, 2002]. Figure 3.4 displays the process for reading or writing to an atomic object.
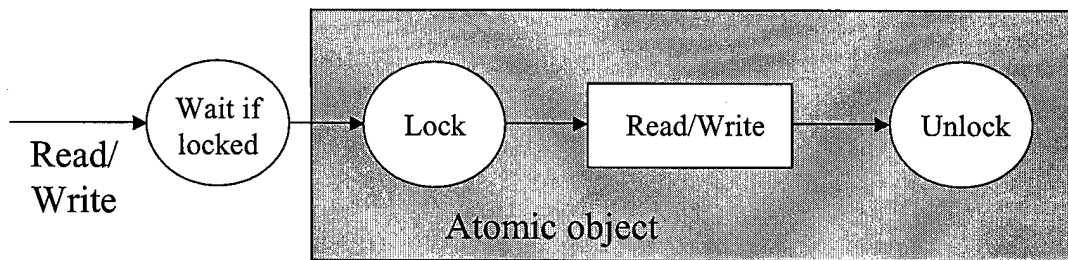
Figure 3.4 Read/Write to atomic object.

Another facility provided by GRRDE is time synchronization. This synchronizes processor clocks on each computer. This is necessary since there can be drift between the clocks on different boards and a calculation on one board could depend on a time-value sent from another.

Functions for performing endian conversion are also provided. This is necessary because the embedded hardware is big endian, while the PCs used to visualize simulations are little endian.

3.4.1 Contracts

Interprocess communication is extended with contracts. These are agreements between two processes to deliver information from one process to the other. This delivery can be periodic or aperiodic. Periodic contracts result in the delivery of information at a set rate and are best suited to continuously varying state information. Aperiodic contracts, which result in the delivery of information only when the associated variable changes, are useful for infrequently changing information.

Central to the concept of contracts is the notion of the dispatch function. A dispatch function fills up a signal with the relevant information and sends it to the destination process. Each contract is associated with a specific dispatch function when the contract is created. Contracts can be set up by the destination process ("pull" contract) or the source process ("push" contract).

Several parameters are specified when creating a contract. We must identify the desired dispatch function, the source and sink processes, the period, the activation time (ie. there can be a delay before starting the contract) and the contract duration.

Two GRRDE processes manage contract creation and execution. These are the message negotiator and message dispatcher, respectively. Contract requests, modifications, and
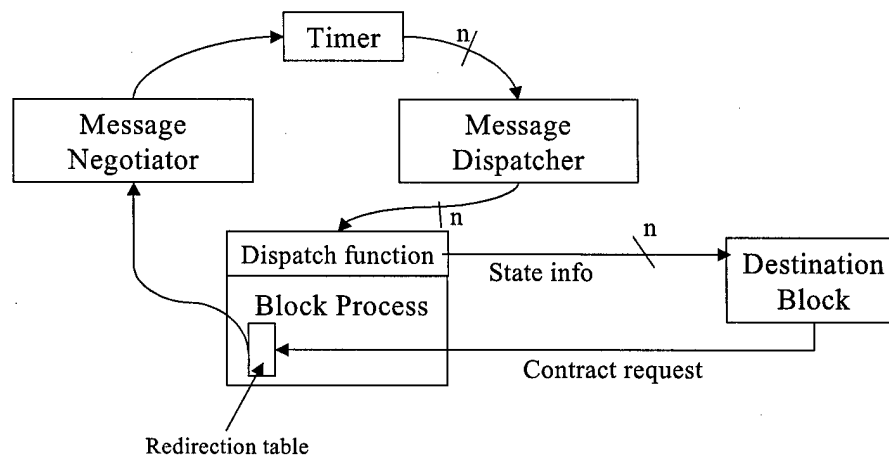
Figure 3.5 Periodic contract setup and dispatching.

service availability queries are sent to the message negotiator of the source block. The message dispatcher handles the execution of contracts by calling the dispatch function at the appropriate times. For periodic contracts, a timer notifies the message dispatcher to dispatch the contract at the correct times. The full sequence is shown in Figure 3.5. Dispatching of aperiodic contracts is handled differently, as evidenced in Figure 3.6. For
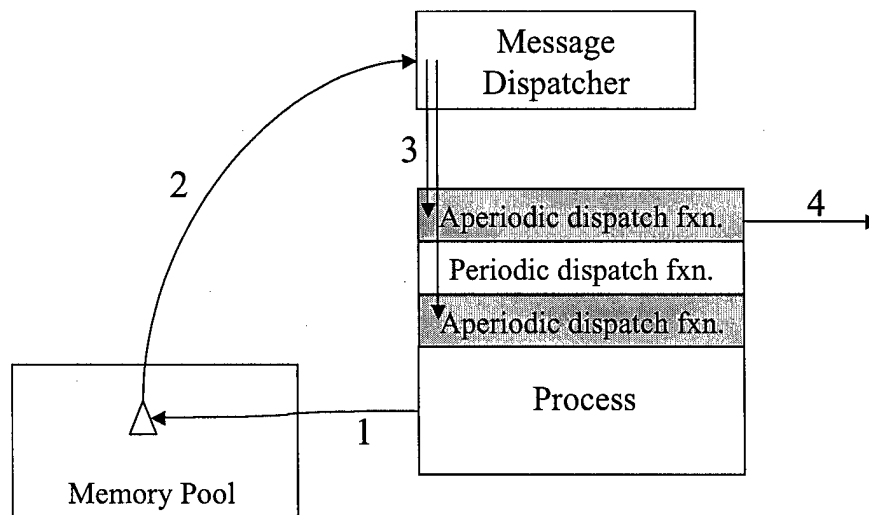
Figure 3.6 Dispatching of aperiodic contracts.

these, the relevant variables take the form of flagged atomic objects. This special type of atomic object has a flag that gets toggled between zero and one whenever it is written to (1). The object then sends a signal to the message dispatcher to notify it that one of the flagged atomic objects has changed value (2). The message dispatcher does not know which object has changed, or which dispatch function accesses that object, so it calls all aperiodic dispatch functions (3). By comparing the flags of the atomic objects that it references with their values at the last dispatch, each dispatch function can determine whether information has changed and, if so, send the information (4).

Contracts simplify simulation development by separating the distribution of information from its creation. This allows for easier design, faster implementation, cleaner code and quicker debugging.

3.4.2 GRRDE Module Structure

By organizing communication between modules into a common framework, GRRDE allows for easier and more efficient simulation module development. User processes contain the code that accomplishes the desired tasks set forth by the developer when conceiv-

ing the module. They are supported by a number of GRRDE processes. Some of these processes are provided, while others that must be present have to be coded by the developer. The message negotiator and message dispatcher, explained above, are provided automatically and are not modified by the programmer. Two processes that must be coded by the developer, but that play specific roles within the GRRDE module framework, are the block manager and input arbiter. The block manager is the only process that is automatically started when a module is run. It can be used to start the other processes, set up or reset contracts, initialize global variables, and look up other processes or blocks. The input arbiter receives signals and either routes them to the appropriate user process, or extracts the data from the signal and saves it in the appropriate global variable. Finally, the block process, which is automatically created, redirects GRRDE signals that it receives to GRRDE processes within the block. For example, new contract requests and contract modification requests are redirected to the message negotiator. All signals that are not specified in the table are redirected to the input arbiter.

3.5 Summary

This chapter provided a quick introduction to GFLOPS. It began with a description of the hardware, then went into the software in greater detail. The main features of OSE, the real-time operating system used for GFLOPS, were explained. Then, the GFLOPS Rapid Real-Time Development Environment was addressed, and the ways in which it simplifies interprocess communication and program organization were discussed. The topics discussed in this chapter help to understand Chapter 4, which details the design of the simulation modules.

Chapter 4

SIMULATOR ARCHITECTURE AND MODULES

4.1 Introduction

This chapter describes the GFLOPS SPHERES Simulator in detail. It begins with some insight into the objectives that drove the design of the simulator. It then discusses each of the three main simulation modules in turn. These are the dynamics, metrology and thruster simulators. For each of these, the functions that the module performs are explained and the interactions with other modules are outlined. Next, the SPHERES flight software module is covered. We see how the SPHERES interrupts were converted into OSE processes and we learn of the changes to SPHERES code that were necessary to allow it to run in the GFLOPS environment. Two applications that allow the user to monitor simulations as they proceed are dealt with next. These are the 3-D viewer and a program that simulates the user interface of the SPHERES laptop. The communications manager is a module that can be used to filter the communications between the SPHERES and the simulated laptop, which is necessary since there is a limited bandwidth. Finally, methods for investigating the resource usage of the SPHERES code, both in terms of processor utilization and memory usage, are discussed.

4.2 Design Objectives

Maximizing the usefulness of the GFLOPS SPHERES Simulator required some high-level objectives. First, as much as possible, the simulator had to be adaptable to

SPHERES hardware changes. The primary reason for this was that the final flight hardware was not fully designed when the simulator was developed. Therefore, the simulator could not be constrained to precise hardware specifications. This adaptability was afforded by placing constants that describe the hardware (eg. force output of a thruster), into a single file, instead of hard-coding these constants into the logic of the modules. Thus, as the new specifications become available, they can simply replace the old ones in the constants file.

The simulator also had to remain adaptable to changes of the SPHERES software since the SPHERES flight code will be modified frequently, even in between tests onboard the International Space Station. In order to achieve this objective, changes needed to adapt pre-existing SPHERES code to the GFLOPS testbed (eg. for communications between satellites) had to be implemented in a way that would be transparent at the level of the guest scientist control code. That is, we wanted to have to replace code with OSE function calls only in the lowest-level SPHERES functions. Then, when the Standard Control Interface is used, the same guest scientist algorithm can be compiled successfully for the GSS or for the actual SPHERES hardware. The guest scientist need not be aware of the contents of the low-level functions.

Another design objective was to make sure that the simulator was capable of handling any control architecture (eg. leader-follower, master/slave). The simulator modules (the thruster, dynamics and metrology simulators) were designed in a very generic way and do not limit this flexibility. The main requirement for achieving different formation flying architectures was to allow for communications between the satellites themselves, and between the satellites and the simulated laptop.

Modularity was also considered a very desirable trait. Separating functions such as thruster, dynamics and metrology simulation into different modules increased the likelihood that changes or improvements to the simulator would be constrained to a small section of code. For example, if we wanted to improve the modelling of thrusters, then we

would only need to modify the thruster simulator. This increased the maintainability of the simulator. Thorough commenting and documentation was also carried out in order to facilitate maintenance by others.

Other objectives included closely maintaining the timing characteristics of the flight code and making it possible to investigate flight code processor utilization. In addition, we wished to utilize the GRRDE toolset to simplify and speed up development of the simulator.

Figure 4.1 shows the general architecture of the GFLOPS SPHERES Simulator. Details
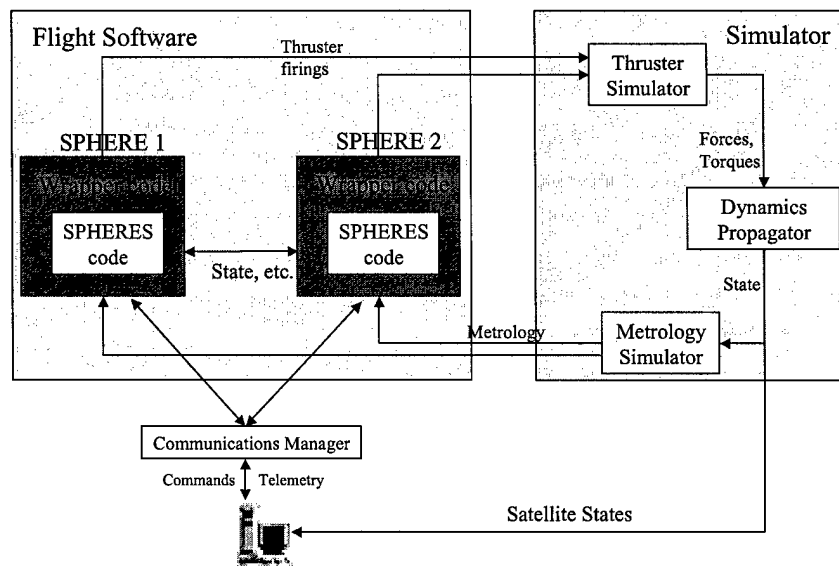
Figure 4.1   Simulation Architecture.

about each of the elements of the simulator are given in the following sections.

4.3 Dynamics Simulator

The dynamics simulator's primary function is to propagates the state of the SPHERES. It receives forces and torques from the thruster simulator and provides state information to

the metrology simulator. It can handle collisions between two SPHERES, or between a SPHERE and a wall. It can also accomodate docking between two SPHERES, where the units stick together after the dock. In addition, the user can choose to log a history of the forces and torques acting on a SPHERE, or he can apply a disturbance force or torque to the SPHERE. These capabilities will now be described in more detail.

4.3.1 State Propagation

The dynamics simulator propagates the states of all satellites. Thirteen variables are propagated for each SPHERE: three for position, three for velocity, four that make up the quaternion that describes the satellite's orientation in the global frame, and three for rotational velocity about each of the satellites body axes. The SPHERE's acceleration and angular acceleration are also contained in six other variables; these are simply updated when thrusters are turned on or off.

To solve the equations of dynamics of the SPHERE, a public domain function for the numerical solution of systems of first order ordinary differential equations with initial conditions is used. Named *Dverk*, it employs a Runge-Kutta algorithm based on Verner's fifth and sixth order pair of formulae and attempts to keep the global error proportional to a specified tolerance. To tailor this solver to the problem at hand, the relevant first order equations had to be specified. The equations for position and velocity are as follows:

$$\dot{\bar{r}} = \bar{v} \tag{4.1}$$

$$\dot{\bar{v}} = \frac{\bar{F}}{m} \tag{4.2}$$

where m is the mass of the SPHERES unit and $\bar{F}$ is the force exerted on it. Inside the function that contains the first order equations, and which is called repeatedly by *Dverk* with different values for the time argument, the force on the SPHERE is converted from the body frame to the global frame. This ensures that the global frame force values are

kept accurate, which is important because if the SPHERE is spinning, then the force vector, while constant in the body frame, is rotating in the global frame.

The rates of change of the quaternion elements are:

$$\frac{d}{dt}\begin{bmatrix} \varepsilon_x \\ \varepsilon_y \\ \varepsilon_z \end{bmatrix} = \frac{1}{2}\begin{bmatrix} \eta & -\varepsilon_z & \varepsilon_y \\ \varepsilon_z & \eta & -\varepsilon_x \\ -\varepsilon_y & \varepsilon_x & \eta \end{bmatrix}\begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \qquad (4.3)$$

$$\dot{\eta} = -\frac{1}{2}(\bar{\varepsilon} \cdot \bar{\omega}) \qquad (4.4)$$

The rate of change of angular velocity about the body axes are:

$$\dot{\bar{\omega}} = (I)^{-1}(\bar{T} - \bar{\omega} \times (I\bar{\omega})) \qquad (4.5)$$

where $\bar{\omega}$ is the angular velocity vector, I is the inertia matrix of the satellite and $\bar{T}$ represents the torques about the body axes. The forces and torques acting on the SPHERE are sent to the dynamics simulator from the thruster simulator. They will be further discussed in Section 4.5.

The SPHERE's state must be re-propagated each time the force or torque acting on it changes.. If thruster states were constant for long periods of time (eg. a second), the most computationally efficient method of updating the state would be to propagate the dynamics far into the future, and use an interpolation function to obtain intermediate values. However, since the SPHERE's control frequency·is 10 Hz and different thrusters turn off at different times, we would be repropagating frequently anyway, so there would not be much point in interpolating between propagations (it would only complicate the code). Therefore, we do not use interpolation, and we propagate the state up to the current time whenever the thruster states change. In addition, the state is automatically propgagated if the thruster states haven't changed in the last 5 ms. Given the relatively slow motion of a

SPHERE, this update rate is sufficient to achieve accurate metrology simulation. Thus, the state is always between 0 and 5 ms old.

4.3.2 Dynamics Simulator Features

Arbitrary Offset of SPHERE Center of Mass

A feature of the dynamics simulator is that an arbitrary offset of the SPHERE's center of mass from its geometric center can be specified. We should note that it is the position and velocity of the center of mass that get propagated. Thus, since the SPHERE requires knowledge of the position and velocity of its geometric center, conversions are made before sending values to the metrology simulator, as follows:

$$\bar{r} = \bar{r}_{cm} + \bar{r}' \tag{4.6}$$

$$\bar{v} = \bar{v}_{cm} + \bar{\omega} \times \bar{r}' + \bar{v}' = \bar{v}_{cm} + \bar{\omega} \times \bar{r}' \tag{4.7}$$

$$\bar{a} = \bar{a}_{cm} + \dot{\bar{\omega}} \times \bar{r}' + \bar{\omega} \times (\bar{\omega} \times \bar{r}') + 2\bar{\omega} \times \bar{v}' + \bar{a}' = \bar{a}_{cm} + \dot{\bar{\omega}} \times \bar{r}' + \bar{\omega} \times (\bar{\omega} \times \bar{r}') \tag{4.8}$$

where the subscript *cm* refers to the state of the center of mass of the SPHERE, and $\bar{r}'$, $\bar{v}'$ and $\bar{a}'$ refer to the relative position, velocity and acceleration of the SPHERE's geometric center with respect to its center of mass, in the body frame ($\bar{v}'$ and $\bar{a}'$ are zero).

Collisions

Collisions are checked for every 100 ms. By approximating their shape as spherical, we can test for collisions between two units by checking whether the difference in their positions is less than two radii of a SPHERE. When a collision occurs, the SPHERES velocities are changed to reflect this. The magnitude of the component of the relative velocity of approach that is parallel to the surface normal at the point of collision is multiplied by the coefficient of restitution for two SPHERES, e_S, to obtain the magnitude of the relative velocity of separation:

$$\left| \frac{(\bar{v}_{1f} \cdot \hat{n}) - (\bar{v}_{2f} \cdot \hat{n})}{(\bar{v}_{1i} \cdot \hat{n}) - (\bar{v}_{2i} \cdot \hat{n})} \right| = e_s \qquad (4.9)$$

Note that e_s is chosen arbitrarily. Here, $\hat{n}$ is a normal at the collision surface, and the subscripts i and f denote conditions just prior to and just after the collision. The normal is an approximation derived from the spherical approximation. Since the satellites have the same mass, their changes in velocity will be equal in magnitude and opposite in direction in order to conserve momentum. We do not capture rotational effects during collisions (ie. angular momentum is not conserved). In addition to SPHERE-SPHERE collisions, a check can be made to see whether the satellite has collided with a wall surrounding the test space. If a collision has occurred, the component of the unit's velocity that is perpendicular to the wall is multiplied by $-e_w$, where e_w is the coefficient of restitution for a collision between a SPHERE and a wall. This simulates a bounce off of the wall with some energy loss.

Docking

If the user wishes to run an algorithm that tests docking between two SPHERES, the dynamics simulator can be compiled to support docking. The user sets a vector that specifies the direction in body coordinates that points from the center of the SPHERE to the middle of the docking port or panel (currently assumed to be the same vector for all SPHERES). When a collision occurs between two SPHERES, the dynamics simulator checks to see if the docking ports for the two SPHERES have come into contact. This is done by checking the orientation of the two units and the proximity of their docking ports. For them to have docked, the vectors pointing to their docking ports must be pointing in opposite directions and their docking ports must be in close proximity. These relations are expressed in the following inequalities,

$$\mathrm{acos}((R_1 \hat{\alpha}) \cdot (R_2 \hat{\alpha})) > \pi - \Phi \qquad (4.10)$$

$$\left\| (\bar{r}_1 + \rho R_1 \hat{\alpha}) - (\bar{r}_2 + \rho R_2 \hat{\alpha}) \right\| < \zeta \tag{4.11}$$

where R_i is the rotation from SPHERE i's body frame coordinates to global frame coordinates, $\hat{\alpha}$ is the unit vector pointing towards the docking port in body coordinates, Φ is an angle representing the maximum angular offset of the docking ports, $\bar{r}_i$ is the position of SPHERE i, ρ is the radius of a SPHERE, and ζ is the maximum linear port offset.

When the above constraints are met, the SPHERES are considered to have docked. Since it is assumed that the docking port will consist of velcro, once two SPHERES have docked they remain together as a rigid body. The center of mass of the composite object, $\bar{r}_{ccm}$, is the average of the centers of mass of the two SPHERES, while conservation of momentum gives the velocity of the combined center of mass, $\bar{v}_{ccm}$:

$$\bar{r}_{ccm} = \frac{\bar{r}_{cm1} + \bar{r}_{cm2}}{2} \tag{4.12}$$

$$\bar{v}_{ccm} = \frac{\bar{v}_{cm1} + \bar{v}_{cm2}}{2} \tag{4.13}$$

Angular momentum must also be conserved during the docking. This is done by considering the angular momentum about the location of the composite center of mass. Just prior to the docking, this will be equal to the following (the angular momentum resulting from the rotation of the units is ignored before the docking):

$$\bar{H}_{ccm(i)} = \bar{r}_{cm1} \times m\bar{v}_{cm1} + \bar{r}_{cm2} \times m\bar{v}_{cm2} \tag{4.14}$$

where now $\bar{r}_{cmi}$ is the position of the SPHERE i's center of mass with respect to the composite center of mass.

After the docking, the angular momentum will be due to the rotation of the composite object about its center of mass:

$$\overline{H}_{ccm(f)} = I_c \overline{\omega} = \overline{H}_{ccm(i)} \tag{4.15}$$

where I_c is the inertia matrix of the composite object. The angular rates of the composite object can be arrived at by multiplying both sides of equation 4.15 by I_c^{-1}. Since the relative orientations of the docked SPHERES cannot be known a priori, I_c must be determined at the time of the docking by combining the moments of inertia of each of the SPHERES. First, using the parallel axis theorem, each of their moments are determined about the composite object's center of mass, shown here for SPHERE i:

$$\boldsymbol{I}_i = \boldsymbol{I}_i + m(\overline{r}^T_{cmi}\overline{r}_{cmi}\mathbf{1} - \overline{r}_{cmi}\overline{r}^T_{cmi}) \tag{4.16}$$

Next, they can be combined together after rotating them into the composite object's body frame:

$$I_c = R_1 I_1 R_1^{-1} + R_2 I_2 R_2^{-1} \tag{4.17}$$

Once docking has occured, the simulator propagates the position, velocity, orientation, and angular velocity of the composite object. Since the positions of the docked SPHERES with respect to the combined center of mass and their orientations with respect to that of the composite object are known, the states of the SPHERES themselves can be calculated with respect to the global frame. For position, velocity and acceleration, modified versions of equations 4.6, 4.7, and 4.8 are used, with the center of mass quantities replaced with combined center of mass values:

$$\overline{r} = \overline{r}_{ccm} + \overline{r}' \tag{4.18}$$

$$\overline{v} = \overline{v}_{ccm} + \overline{\omega} \times \overline{r}' + \overline{v}' = \overline{v}_{ccm} + \overline{\omega} \times \overline{r}' \tag{4.19}$$

$$\overline{a} = \overline{a}_{ccm} + \dot{\overline{\omega}} \times \overline{r}' + \overline{\omega} \times (\overline{\omega} \times \overline{r}') + 2\overline{\omega} \times \overline{v}' + \overline{a}' = \overline{a}_{ccm} + \dot{\overline{\omega}} \times \overline{r}' + \overline{\omega} \times (\overline{\omega} \times \overline{r}') \tag{4.20}$$

where $\vec{r}'$ is now the original offset of the SPHERE's geometric center from its center of mass plus the offset of the SPHERE's center of mass from the combined center of mass. Because the docked SPHERES form a rigid body, the angular rate and acceleration can be arrived at by simply rotating those of the composite object into the SPHERE'S body frame.

Immediately after executing the dock, the dynamics simulator will send a signal to the thruster simulator to notify it that docking has taken place. This is necessary because the thruster simulator must recalculate the positions of thrusters with respect to the new center of mass, the directions in which thrusters produce force in the new body frame, and the torque produced by a unit force from a thruster. During the time between docking and acknowledgment from the thruster simulator that these new values have been calculated, the dynamics simulator ignores forces and torques sent from the thruster simulator. This is to avoid having the docked SPHERES fly apart.

Force and Torque Recording

The dynamics simulator is capable of sending signals containing a log of the commanded forces and torques acting on a SPHERE to other applications. Forces are specified in the global frame, while the torques are expressed in the body frame. Each time a force or torque is received from the thruster simulator, the data are saved and time-stamped. To avoid excessive signal traffic, data are accumulated until they occupy the maximum OSE signal size. Then the signal is sent to the requesting application.

User-Applied Disturbances

The dynamics simulator can apply arbitrary force and torque disturbances to the SPHERES. The duration of the disturbance is specified in milliseconds. This time will be rounded up to the next integral number of integration periods. Thus, the applied disturbance may persist for slightly longer than requested. The disturbance force and torque, their duration, and the ID of the affected SPHERE are sent to the dynamics simulator in a signal. Note that the magnitude and direction of the force or torque is constant for the

entire duration of the disturbance. More complex disturbances would need to be coded into the state propagation routines directly.

4.3.3 External Signals

The dynamics simulator receives several external signals, mostly commands from the user or other modules:

- **DYN_SIM_FORCE_TORQUE_INPUT** is sent by the thruster simulator and contains the updated torques and forces acting on a SPHERE and that SPHERE's ID.

- **DYN_SIM_SET_INITIAL_STATE** contains the thirteen variables that make up the initial state of the SPHERE and is sent to the dynamics simulator when a new satellite is added to the simulation.

- **START_SIMULATION** starts the simulation by informing the dynamics simulator to start propagating the satellites' states.

- **DYN_SIM_DISTURB_SPHERE** causes a SPHERE to experience a disturbance force or torque for a specified length of time.

- **DYN_SIM_REQ_THRUST_STATS** is sent by applications that wish to receive records of the force and torque history for all active SPHERES.

- **DYN_SIM_STOP_THRUST_STATS** stops the sending of force/torque logs.

- **DYN_SIM_RESET_SIM** resets the states of all SPHERES to their default values and removes them from the simulation.

- **DYN_SIM_DOCKING_ACKNOWLEDGE** is received from the thruster simulator and signifies that the dynamics simulator no longer needs to ignore force/torque signals sent for recently docked SPHERES.

Three dispatch functions are provided by the dynamics simulator:

- **dyn_sim_full_state** sends a signal containing the values of the thirteen state variables for the requested satellite.

- **dyn_sim_extended_state** sends the same information as dyn_sim_full_state along with the six acceleration variables.

- **dyn_sim_all_sats_state** is similar to dyn_sim_extended_state, but contains the data for all satellites at once.

Each of these functions also send the time that has elapsed since the start of the simulation.

4.4 Metrology Simulator

4.4.1 Metrology Simulation

The metrology simulator enables the SPHERES to receive Inertial Measurement Unit (IMU) and global metrology readings. IMU readings contain the values from the three single-axis rate gyros and the three-axis accelerometer. As soon as a SPHERE receives an IMU reading, it asks for a new one. To ensure that the time between IMU readings is not too small, the metrology simulator waits for 18 ms before fulfilling an IMU request (it is not known what the actual time between received IMU measurements is for the SPHERES hardware). The IMU readings sent to the SPHERE are in units of millivolts.

The metrology simulator models the non-ideal characteristics of the accelerometer and gyros. These parameters were specified in Table 2.3 and Table 2.4 respectively. If the acceleration or angular rate is outside of the input range, the measurement saturates at the extremity of the range. The accelerometer resolution is also modeled, so that the output of the accelerometer can only be a multiple of 5 μg.

Noise is added according to the values given in the tables, with the assumption that no noise exists outside of the bandwidths listed. The noise entries in the tables are given as square roots of power spectral densities. These can be used to determine the noise measured in bandwidth B as follows [Fish, 1994]:

$$e_{RMS} = \varepsilon B^{\frac{1}{2}} \tag{4.21}$$

where ε is the root of the power spectral density in bandwidth B. The root mean square value of the noise is e_{RMS}. Noise is added to the signals using a GRRDE normal random number generator with a mean of zero and a stardard deviation of e_{RMS}. The resulting values, in rad/s or m/s^2, are converted to millivolts according to the following equation:

$$V = (S + e)(ScaleFactor) + Bias \tag{4.22}$$

where V is the resulting voltage, S is the value of acceleration or angular rate received from the dynamics simulator, e is the noise added to the signal, *ScaleFactor* is the scale factor of the device in millivolts/unit, and *Bias* is the bias of the device in millivolts.

Currently, the metrology simulator is compliant with the global metrology system for the prototype SPHERES. The metrology simulator starts a new global metrology transmit cycle at a set period, nominally every 153 ms. In contrast, with the flight SPHERES, the "master" SPHERE will create an infrared flash to request a new global metrology cycle. With the real prototype SPHERES, the CPU would request a global metrology reading by notifying the metrology board (the Tattletale 8), which would return the data once it has been acquired. In the simulation, instead of communicating with the Tattletale, the SPHERE requests global metrology data from the metrology simulator. At the start of each cycle, the metrology simulator checks which SPHERES units have requested global metrology data. After the initial 5 ms delay (see Section 2.4.4), the distances from each transmitter to each receiver on the SPHERE are computed and saved. This is done for a different transmitter every 20 ms. After all beacons have been considered, the completed measurement is sent to the SPHERE. These delays make the simulated measurements and timing representative of the actual system.

The distance measurements must be modified to account for the actual behavior of the hardware. As explained in Section 2.4.4, the times-of-flight measured by the global metrology system depend on the distance and relative orientation between transmitter and receiver. A modified version of the function `correct_gGlobal` from the SPHERES flight software is called from the metrology simulator to apply these corrections. The function is the same as in the flight software, except that the opposite corrections are applied to simulate the physical effects that are corrected in the flight software. With this reverse correction applied by the metrology simulator, the distances are expected to be close to those that would be measured in the real system. Of course, these values can only be as accurate as the calibration that was used to create the `correct_gGlobal` function. However, even with a perfect calibration, the distances that the SPHERE calculates

after calling **correct_gGlobal** would not be perfect. In order to perform the correction, the SPHERE needs to estimate the transmitter angle (θ_t in Figure 4.2) and the
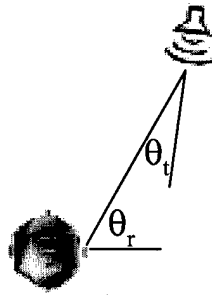
Figure 4.2 Receiver and transmitter angle.

receiver angle, θ_r, for each beacon-receiver pair. The SPHERE estimates these angles using the uncorrected distance measurements, so there will always be some error in the angles that are arrived at, and hence in the distance correction.

4.4.2 External Signals

The metrology simulator receives several external signals:

- **DYN_SIM_ALL_SATS_STATE** contains the state updates from the dynamics simulator. These are sent every millisecond. Because the dynamics simulator can take as long as 5 ms to update the state for a satellite, consecutive signals often contain duplicate information. This is not a problem.

- **SPH_SENSOR_SIM_GM_REQUEST** contains a global metrology request form a SPHERE.

- **SPH_SENSOR_SIM_IMU_REQUEST** contains an IMU request from a SPHERE.

4.5 Thruster Simulator

The thruster simulator module is sent THRUST_SIG signals from the SPHERES in the simulation every time they change the combination of thrusters that are activated. The

THRUST_SIG signal contains a bit-packed integer that indicates whether each thruster is on or off. The simulator models the force profile of a thruster as shown in Figure 4.3.
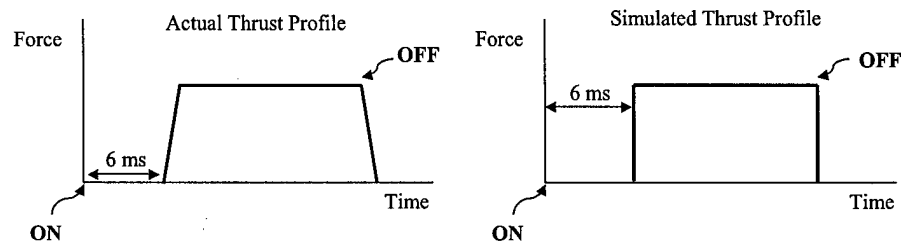
Figure 4.3   Actual and simulated thrust profiles.

A thruster takes 6 ms to open. This is due to the fact that voltage must be applied for this minimum amount of time before the the valve will begin opening.   Once the valve starts to open, it takes approximately 1.1 ms[1] for the force to reach its nominal value of 0.2 N. While this transient behavior is depicted in the left side of the figure as a linear increase in thrust, it is in fact non-linear. There is also some transient behavior as the valve closes. To simplify the representation of the thrust profile, the simulated profile ignores the transients. After the 6 ms delay, the force takes its nominal value, and it goes immediately to zero when a thruster is commanded off. For the linear profile in the left side of the figure, if the ramp up and ramp down time are equal, then the right profile will have the same area under the curve, or same impulse. The thruster simulator keeps a record of the last commanded thruster states for each satellite, the time of these commands, and the current thruster states. During the 6 ms opening delay, the current and commanded states will differ.

The main thruster simulator process runs every millisecond and performs a variety of tasks. First, it checks for unprocessed THRUST_SIG signals waiting in its signal queue. It processes them all and updates the record of current thruster states. Second, the simula-

---

1. This value of 1.1 ms was obtained by Simon Nolet, SPHERES team member and MIT SSL graduate student, in email correspondence with hardware manufacturers.

tor checks whether the valve opening delay has just expired for any thrusters and, if so, records the change. If the thruster states have changed for a SPHERE, either from a new command or from a delay expiring, new force and torque values are sent to the dynamics simulator. Normally distributed noise can be added to the output of a thruster. The forces about the center of mass in the body frame are computed as follows:

$$\bar{F} = \sum_{i=1}^{N} (Th_i)\hat{d}_i \qquad (4.23)$$

In this equation, N is the number of thrusters per SPHERE, Th_i is the current thrust output of thruster i, and $\hat{d}_i$ is a unit thrust in the body frame.

Since the thruster positions and the thrust vectors are known, we can compute the torques about the body axes that arise from a unit force from each thruster:

$$\bar{T}_i = \bar{R}_i \times \hat{d}_i \qquad (4.24)$$

Here, $\bar{T}_i$ is the torque that results when thruster i produces a unit force, while $\bar{R}_i$ is the position of thruster i with respect to the center of mass of the SPHERE. With N thrusters, the total torque is given by:

$$\bar{T} = \sum_{i=1}^{N} (Th_i)\bar{T}_i \qquad (4.25)$$

When a docking between SPHERES is sensed by the dynamics simulator, it sends a signal to notify the thruster simulator. The signal contains the IDs of the affected units, the positions of their geometric centers with respect to the new center of mass, and the quaternions representing their orientations in the new body frame. The thruster simulator recalculates equations 4.23 and 4.24 so that thruster firings for each of the docked SPHERES are now converted into forces and torques specified in the new body frame. And when sending the

forces and torques of one of the docked SPHERES to the dynamics simulator, the thruster simulator adds the values that are due to the thrusters from both docked SPHERES.

4.6 SPHERE Module

The SPHERE module encapsulates the SPHERES flight code. The primary goal in designing this module was to minimize differences between simulation code and actual flight code. However, the SPHERE module is running on different hardware and a different operating system than in the actual system, so it was impossible to make the code identical. Low-level calls that access hardware could not be left unchanged in the simulation. In particular, instead of using wireless communications to communicate with other units or the laptop application, the SPHERE module must communicate using OSE signals. However, the goal of the GSS was not to test the communications hardware. It was meant to verify the guest investigator formation flying algorithms. These will almost certainly contain satellite-to-satellite (STS) and satellite-to-laptop (STL) communications, but as long as the information arrives at the destination, we need not be concerned with the exact transmission method. Furthermore, through careful design of the SPHERE module, we ensured that the timing characteristics of communications and other functions are similar to those in the real system.

4.6.1 SPHERE Module Processes

As explained in Section 2.4.2, the SPHERES flight code is partitioned into four principal sections. There are two timer-interrupts (the thruster actuator and controller), an event-driven interrupt (for communications) and some background processing. Correspondingly, there are four main elements of the SPHERE module. At first glance, the logical choice would have been to replace the thruster and controller sections with OSE timer-interrupt processes, and the communications section with an OSE software interrupt process that gets awakened when it receives a signal. However, interrupt processes in OSE cannot send or receive signals from processes that are not part of the same block. Since the thruster actuator must send thrust signals to the thruster simulator, and the communica-

tions interrupt must communicate with several other modules, these could not be coded as OSE timer-interrupts. Furthermore, interrupts have higher priority than any other process in OSE. Thus, the controller, which must be at a lower priority than the other two, could not exist as a timer-interrupt either.

Thruster Actuator

Instead of an interrupt-based solution, the thruster actuator was implemented as a dispatch function named "SPHERE_send_thrusts". The dispatch function increments counters that keep track of time, propellant usage and battery usage, asks for a new global metrology reading every second, and checks whether each thruster should be on or off. The code for the dispatch function is taken straight from the original code for the propulsion interrupt. The only substantial difference is that, instead of writing the thruster commands to a hardware port, it writes them to a signal that gets sent to the thruster simulator. A push contract between the thruster actuator and the thruster simulator ensures that thrust values can be updated every millisecond, just as in the flight code. However, the simulator is only notified when the thruster states change. If the thruster settings remain constant, we do not send updates to the simulator.

Communications Process

The communications interrupt routine inputs received communications data into global arrays so that the data can be accessed by other processes. It was coded as a prioritized process that is enabled whenever it receives a signal, just as the actual communications interrupt wakes up whenever incoming data is available. Minor adaptations were made to the code. For example, instead of reading from a hardware device, communications messages are received via OSE signals. Nonetheless, the data gets put in the same arrays and the timing characteristics are comparable.

Controller

The controller process is extremely simple. It starts a timer that sends stimulus signals to the process at 50 Hz. Each time this wrapper process receives one of these signals, it executes the flight version of the controller code.

Background Processing

The background processing from the flight code was put into an OSE background process. This code receives data packets from the ground laptop, transmits packets to the ground and other satellites, and calls the state determination and housekeeping routines. Some of the commands that can be sent to a SPHERE do not make sense in the simulation. For example, a watchdog timer in the flight code periodically checks to see if the processor is still running. If not, it will reset the processor. This capability is not reflected in the simulation since is not needed to evaluate guest scientist code. Therefore, there is no RESET command that causes the watchdog timer to automatically reset the processor, even though there is in the actual SPHERES system.

4.6.2 Communications

While the physical communications links are different from the operational system, the communications protocols are not. The only difference between simulation and flight systems is that the low-level reads and writes to hardware were replaced with the receiving or sending of signals. The formatting of messages into packets for transmission as single bytes remains unchanged. This includes the calculation of checksums and the use of a token ring protocol. While there is no real need to send a message as several bytes in the simulation, or to compute checksums since there will be no bit errors (although this could be simulated), it is desirable to do so since it keeps the timing behavior similar. If messages were sent as a single signal, the relative processing time between the four software sections could change. Furthermore, confining adaptations to a few low-level functions ensured that few changes are necessary when flight code is updated. Another characteristic of communications that was represented is that when one SPHERE sends a packet, all

other satellites will receive it and must determine from the packet header whether it was meant for them. This maintains the same relative communications processing loads between separate satellites.

4.6.3 Modifications to SPHERES Code

A useful feature of the simulator is the capability of providing the flight code with perfect knowledge of the SPHERE's state. This can be accomplished by receiving the state directly from the dynamics simulator, bypassing the metrology simulator. This allows us to investigate the best-case performance of formation flying algorithms. Perfect state knowledge eliminates the ambiguity between poor performance due to a flawed algorithm and due to sensor limitations.

Other modifications to flight code did not offer any benefits, but were unavoidable to allow for compatibility with the GSS. First, in the SPHERES flight code, the programmer must explicitly instruct the operating system when interrupts may be preempted by higher priority interrupts. This is called nesting of interrupts. By default, this feature is disabled in the flight system. However, to ensure accurate thruster pulse-width resolution, the propulsion interrupt must be able to interrupt the controller. Thus, a call to a function called **NEST_INT** is made at the start of the controller interrupt, and a call to **UN_NEST** is made at the end. In the GSS, these calls are unnecessary since nesting of interrupts occurs automatically because the interrupts are actually coded as *prioritized* processes. The message dispatcher, which calls the thruster actuator, runs at a higher priority than the controller, so preemption is automatic. Furthermore, the flight versions of the **NEST_INT** and **UN_NEST** calls contain assembly language routines that are not compatible with GFLOPS. The GFLOPS SPHERES Simulator redefines these functions as "dummy" routines. This enables guest investigator algorithms to be compatible with both the GSS and the SPHERES hardware.

The function that collects metrology data was rewritten. For the prototype SPHERES, the function **tt8_get** collects metrology data from individual bytes sent from the Tattletale 8

processor. However, in the simulation, a metrology reading is returned in full in one signal. This was done to simplify communications between the metrology simulator and the SPHERE module, and to avoid tying the design of the metrology simulator to the SPHERES code. Thus, `tt8_get` was adapted to handle the new data format.

A problem with the SPHERES flight code is that various pointers exist that point to specific parts of memory as defined in the file main.h. These include the addresses of various registers that do not exist in the SPHERE module (eg. for communications ports, flash memory, etc.). If these non-existent registers were written to, the simulation would behave unpredictably and would probably crash. To alleviate this problem, while not modifying references to these registers in the SPHERES flight code, these pointers were reset to point to memory that is dynamically allocated when the SPHERE module starts. The writes to memory can still occur, but they will have no effect other than to change the values held in these dummy registers. Many of these registers control secondary systems such as LED indicators, so accurate implementation is not required. Using dummy registers allows us to minimize modifications to guest scientist code, ensuring that the flight and simulation code remain compatible.

As mentioned earlier, communications had to be modified to make them compatible with OSE interprocess communications. In particular, the function `send_com(int port, unsigned char out_char)` which writes data to the hardware output register, had to be rewritten. The integer argument specifies the destination: the ground laptop, other SPHERES, or the field programmable gate array (FPGA), while the second argument is a single byte of data. Intersatellite communications are sent to all other satellites via OSE signals, while STL communications are sent to all satellites, as well as the laptop application running on a support PC. For communication with the FPGA, the modified function checks the char argument and sends either a global metrology request to the metrology simulator, an IMU request, or both. The input function `get_com` is unchanged since incoming data is buffered in global arrays by the communications interrupt. The `get_com` function simply accesses these arrays.

Besides the modifications mentioned here, a number of small changes had to be made to various header (.h) files. These modifications were necessary so that some of the flight software files (written in C) could included by other GSS files (written in C++). These changes do not affect the functioning of the SPHERES code in the simulation.

In order to allow the flight code to fit seamlessly into the simulation, some "wrapper" code was needed. For example, the SPHERE needs to search for the process IDs of the communications manager and of other units, so it can send signals to them. The wrapper code also sends the initial state of the unit to the dynamics simulator. Furthermore, the timing and priority of the processes are specified in the wrapper code. The wrapper code contains the elements of the GRRDE module architecture, such as the block manager and input arbiter.

4.7 Communications Manager

The communications manager module facilitates message passing between the SPHERES module and the simulated laptop application. The communications manager exchanges signals with a bridge application (named OSEbridge) running on an OSE soft kernel on the support PC. To complete the final leg of the communications channel, a named pipe is established between OSEbridge and the laptop program. A pipe is a section of shared memory that Windows processes can use to communicate. OSEbridge parses information between the formats needed for these two communications channels. OSEbridge supports two pipes to PC applications, one to the 3-D viewer and another to the simulated laptop command interface. The 3-D viewer draws SPHERE positions using truth data obtained form the dynamics simulator.

OSEbridge also enables the automatic loading and starting of simulation modules upon system start-up. The user puts the names of the software modules to load into a file and OSEbridge sends these names to a process running on the embedded hardware, which then loads and optionally starts the appropriate modules.

The communications manager was needed as an intermediate step between the SPHERE module and OSEbridge due to the limited communications bandwidth of OSEbridge. The OSEbridge must compete for access to the CPU with all of the other PC applications, including the 3-D viewer and the laptop command interface. Running OSEbridge at a higher priority than these other applications resulted in very slow system performance. However, when it was run at the same priority as these other processes, the time that it spent waiting for its next chance to use the CPU was too great to support the real-time communications between the SPHERES and the laptop command window. In testing it was found that OSEbridge would sleep for up to 100 ms. This delay is unnacceptable because when receiving a packet from the ground laptop (or from another SPHERE), the SPHERE has a 4ms second maximum timeout in between bytes, after which it is assumed that there has been a communications error and the rest of the packet is ignored. In order to avoid timeout errors due to the delays experienced by OSEbridge, it was realized that packets should be sent through OSEbridge in full, instead of as individual bytes. However, as already mentioned, a requirement of the simulator architecture was to avoid changes to high-level SPHERES flight code. To preserve this objective and provide reliable communications with the laptop interface, an intermediary node was needed on the communications path. This entity, the communications manager, allows signals to be sent between itself and OSEbridge as whole packets, and between the SPHERE module and itself as single bytes.

The communications manager is made up of two processes, the satellite-to-ground (STG) and the ground-to-satellite (GTS) communications managers. The STG communications manager uses the SPHERES `get_packet` function to collect packets from the individual bytes sent by the SPHERES. Once a full packet is received, it sends it to OSEbridge as one signal. Signals intended for the laptop are forwarded to the other SPHERES, mimicking broadcast communication. The other SPHERES must receive and discard these packets. The GTS communications manager receives packet signals from the laptop, breaks them up into bytes and sends them the SPHERES.

The communications manager filters SPHERES telemetry that is forwarded to the laptop. This filtering is necessary because with several SPHERES in the simulation, the OSE-bridge communications bandwidth gets used up. When the communications manager receives a telemetry packet from a SPHERE, it checks its type (eg. master position, slave angular rate), and compares it to the filtering rules. At compile time, the user can choose to filter (ie. not send) all, none, or some fraction of each type of telemetry message. However, the signals will still get passed on to the other SPHERES (which will discard them since they are meant for the ground laptop). Command and token messages are not filtered. Messages passing in the other direction, from the laptop to the SPHERE, are never filtered.

4.8 Simulation Viewer

Running simulations can be visualized using a 3-D viewer developed for the GFLOPS SPHERES Simulator. The viewer uses OpenGL graphics libraries and shows the motion of SPHERES units in the test space. A screen-shot of the viewer is provided in Figure 4.4. The viewer receives state updates from the dynamics simulator every 100 ms. These are delivered by a standard GRRDE message contract. Thus, it can display the motion of SPHERES at a rate of 10 frames/sec. Lighting helps provide a sense of perspective. Ultrasound beacons are drawn as cylinders with a protruding line denoting their pointing directions. Thus, one can track where the SPHERES are with respect to the test space.

The user can choose from several preset views or he can manually alter the viewpoint. Three modes exist for scene navigation. Zooming moves the viewpoint closer or farther from the image along the line of sight. Flying moves the viewpoint perpendicular to the line of sight. Rotating rotates the viewpoint around the image. Each of these scene navigations are executed with the mouse.

Live simulations can be viewed in real-time. They can also be recorded for later playback by choosing to log the satellite state data to a text file. Thus, with a copy of the viewer and a simulation log, guest investigators can visualize the performance of their algorithms
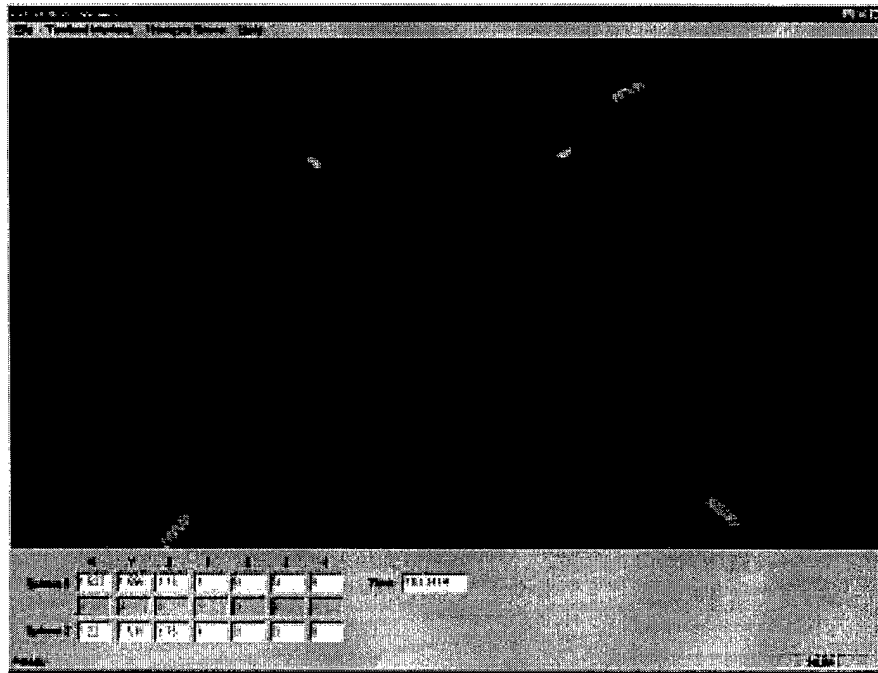
Figure 4.4 SPHERES viewer.

remotely. Because a timer ensures that data is read from the log at the same rate as it was written, playback speed is insensitive to computer performance.

The viewer also allows the user to send disturbance signals to the dynamics simulator, to request that force and torque histories be sent from the dynamics simulator and saved to file, and to reset the simulation.

4.9 Simulated SPHERES Laptop GUI

An application was created to simulate the functioning of the laptop that communicates with the SPHERES. The simulated laptop interface is shown below in Figure 4.5. This application displays telemetry and debug information from the SPHERES and the user can choose from a list of commands to send to a SPHERE. The appearance of this application could be altered to track changes in the flight version. Alternatively, using the code for the
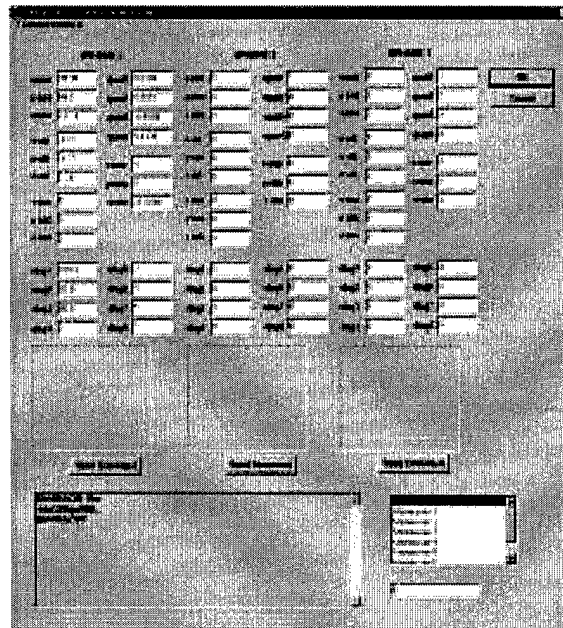
Figure 4.5 Simulated SPHERES Laptop GUI.

simulated GUI as a guide, the final flight GUI could be adapted for communications with the GSS.

The core logic that processes communications to and from SPHERES was adapted from a version of the laptop application that has been used with the prototype hardware. This was done to ensure consistent creation and processing of packets. In particular, because the data received from the units is saved in the same format, the telemetry created during testing on the GFLOPS testbed can be reduced and analyzed using MATLAB scripts created for flight telemetry. Several parts of the application logic required modification. For example, since communications between the laptop and the communications manager use full packets, instead of individual bytes, these differences are reflected in some low-level functions.

4.10 CPU Load Profiler

The OSE Illuminator debugger allows users to measure CPU utilization. A continuously updating graph of load percentages provides a good summary of CPU processing time, as shown in Figure 4.6. Another option is to do a process level load measurement, where the
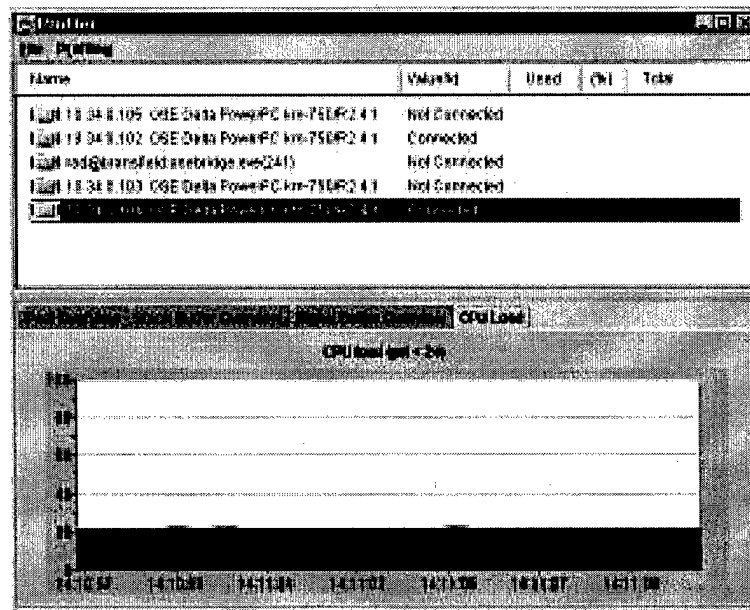
Figure 4.6 CPU load measurement.

load for each process is overlayed on a color-coded graph. The output window for process level load measument is given in Figure 4.7. Alternatively, the data can be displayed and saved as a text listing.

The profiler measures the relative processing needs of the processes in the system. This can be used to analyze the effects of design decisions. The processing requirements of the thruster dispatch function are not expected to vary with different control algorithms, assuming the Standard Control Interface (see Section 2.4.2) is used. The communication process' processing needs will vary with the amount of traffic, but are is not expected to
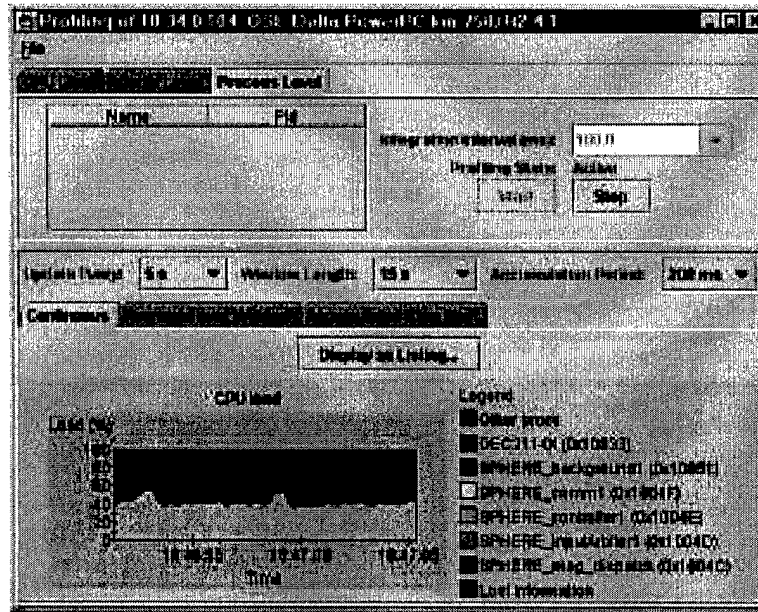
Figure 4.7 Process level load measurement.

make up a significant portion of the processing. The greatest amount of variability is likely to occur with the controller process, since this code will change substantially from test to test. There could also be large differences in background processing requirements. If the Direct Control Interface is employed, the processor utilization of each process could vary significantly for different guest scientist code. With the Custom Control Interface it is not possible to compare one set of code to another, as there could be a completely different set of processes, but we can still observe the absolute load on the processor.

Since the simulator hardware and operating system are not the same as the flight versions, the flight processor utilization may vary from our measurements. In order to draw conclusions about performance on the flight hardware, we need to be able to compare the processors in the two systems. The SPHERES flight hardware features a Texas Instruments digital signal processor, the TMS320C6701, that runs the flight software. Since this is a DSP, it is optimized for vector or parallel computations. For this reason, it has eight functional units that can perform calculations simultaneously [TI, 2000]. The C6701 has four fixed-/floating-point arithmetic logic units (ALUs), two fixed-point ALUs, and two fixed-

/floating-point multipliers. The ALUs perform 32-bit calculations, while the multipliers have 16-bit inputs and 32-bit outputs. Running at 167 MHz, the TMS320C6701 takes 6 ns to perform a cycle. With the six floating-point units operating in parallel, a theoretical maximum performance of 1×10^9 floating-point operations per second (FLOPS) is possible (six operations in 6 ns). However, this maximum performance would only be approached for problems that can exploit the parallel processing throughput of the DSP (ie. those that allow six floating-point calculations to be made in parallel). This is common in DSP applications, where there may be signals coming in from twelve or more channels, but it is not the case for the SPHERES testbed. The majority of calculations will be too simple to break up into six pieces, and the result of one calculation will often be needed before the next can start. For this reason, it is not expected that the theoretical limit of 1×10^9 FLOPS will be approached on the SPHERES hardware.

The GFLOPS testbed relies on a more general purpose processor, the IBM PowerPC 750, running at 400 MHz. This processor has two fixed point units and one floating point unit [IBM, 2001]. The floating point unit is able to perform a single-precision (32-bit) multiply-add operation in one cycle. A multiply-add operation is a ternary operation of the form:

$$a \pm bc \qquad (4.26)$$

Thus, since a multiply-add accounts for two floating point operations, and the processor is running at 400 MHz, the theoretical peak performance of a GFLOPS processor is 800 MFLOPS.

Although GFLOPS will never perform at 800 MFLOPS, it is likely to be closer to its limit than the SPHERES DSP to its own, since the SPHERES processor requires higher parallelism in the computation to make the most of its resources. Despite the lower peak performance of the GFLOPS processors, we expect that they could have a higher effective performance when running SPHERES code. To obtain a quantitative comparison of the speeds of the processors would require measuring the running time of SPHERES code on

both. Even if we cannot draw conclusions from the running time on the GFLOPS hardware, we can expect that the relative processing time of each process will be similar. For example, if the controller process is using most of the processing power in the simulator, we expect to see the same behavior if the code were run on SPHERES.

Another issue that must be considered when attempting to compare processor utilization is the fact that time delays take the same amount of time on any processor. For example, the timeout that occurs when a SPHERE is waiting for a communications byte from another SPHERE that does not arrive (further explored in Section 5.2.2), will take 4 ms no matter what type of processor is being used. In the case of STS communications being received in the controller process, the majority of this time will be counted towards CPU utilization by the controller process (since it only gets interrupted by the thruster and communications interrupts, which do not take much processing time). Hence, we cannot directly convert processor utilization times using a multiplication factor, since the factor would depend on the amount of communications timeouts in the code. One solution would be to estimate the amount of processor utilization that is due to time delays on the GFLOPS system, subtract this amount, then use a factor (assuming we have done a calibration) to predict the utilization due to the rest of the code on the SPHERES hardware, and finally add the timing delays back in.

4.11 Memory Usage

The GSS is also tasked with investigating memory usage by SPHERES code. Flight code must not use more memory than is available from the SPHERES hardware. While the GFLOPS processors have access to 256 MB of RAM, the SPHERES avionics have only 16 MB. Thus, limitting the amount of memory available to the SPHERE module would make the simulator more representative of the target system.

There are several ways that memory is stored for a computer program. Global and static variables are stored for as long as a program runs. Local variables are allocated from the stack when a function is called. The heap is used for dynamic memory allocation (mem-

ory that is allocated only at run-time). SPHERES flight code makes extensive use of each of these types of memory allocation. In addition to storage of program variables, the program execution code must also be stored in RAM

When a process is created in OSE, the size of the stack is specified. Therefore, we can limit the stack memory available to each SPHERES process. However, this is not representative of the situation that actually occurs with the SPHERES flight system. Here, the total memory available is shared between all interrupts.

In OSE, there is one heap for each memory segment. Because all SPHERES processes are part of the same block, they share the same memory segment and therefore the same heap. Furthermore, since the other OSE or GRRDE processes that use the same memory segment do not allocate memory dynamically, all of the memory allocated from the heap belongs to SPHERES processes.

One method for verifying that the SPHERES code does not use more than 16 MB would be to estimate the amount of storage needed for non-dynamic memory, then bound the size of the heap so that the total memory never exceeds 16 MB. Limiting the size of the heap would result in the SPHERE module crashing if the heap fills up. Another option would be to directly track dynamic memory inside the dynamic memory allocation and deallocation functions. The amount of global and static memory could be easily determined. Estimating the amount of stack memory needed would entail summing the maximum requirements for each interrupt. This could be done by determining at which point in each process the most local memory is needed (for all levels of the function call stack). To determine the size of the executable flight code, we could examine the compiled memory image (by getting a member of the SPHERES team to build the code).

4.12 Summary

This chapter gave an detailed account of the GFLOPS SPHERES Simulator. The objectives explained at the start of the chapter helped to understand the reasoning behind many

of the design decisions that were later outlined. The three simulator modules were described with emphasis on their roles, their features, and their interfaces. The SPHERE module was presented next, and care was taken to compare its design and functioning to that of the flight code. The adaptations that had to be made to SPHERES code to allow it to run in the simulator were listed and their effect on the simulator's accuracy were analyzed. It was discussed how the communications manager helps alleviate bandwidth problems for communications between the embedded hardware and the support PC. The 3-D viewer and the simulated laptop, two simulation monitoring applications that run on the PC, were also introduced. Finally, methods for measuring processor load and memory usage were outlined. We saw that determining processor utilization is easily done with tools provided by OSE, while memory estimation mostly has to be done manually by analyzing code.

The next chapter will demonstrate the use of the simulator for a set of representative scenarios. The results discussed in that chapter will allow us to guage the usefulness and accuracy of the simulator.

Chapter 5

SIMULATION RESULTS

5.1 Introduction

This chapter presents the results of several simulations that illustrate the performance of the GFLOPS SPHERES Simulator. A leader-follower simulation demonstrates several of the capabilities of the GSS, including inter-SPHERE communications, CPU utilization measurement, and force recording. A collision test and a passive docking test show the simulator's abilities in these areas. A cooperative docking test on the GSS is used to compare the motion observed with that obtained with the same control code on the SPHERES air table.

5.2 Leader-Follower Simulation

In order to test the intersatellite communications capabilities of the GFLOPS SPHERES Simulator, a simulation scenario was developed that involved a "leader" SPHERE executing a predetermined profile, while a "follower" SPHERE attempted to mimic the motion of the leader (with an offset to avoid collisions). The leader transmitted its state to the follower at a rate of 10 Hz and the follower used this (minus an offset of 30 cm) as its target state. The predetermined profile took the form of a square of 40 cm side length parallel to the XY plane. The leader was commanded to remain at each corner of the square for 10 seconds. No noise was added to either the thruster firings or the metrology readings.

The Standard Control Interface was used for this simulation. The custom code placed in the function **process_maneuverlist** is listed in Appendix B. It should be noted that the function **propagate_state** from the SPHERES flight code, used to update the state every time an IMU measurement is received, was modified to take into account acceleration due to thruster firings. A record is kept listing the time each thruster has been on since the last IMU update. This is used to find the average force exerted on the SPHERE by its thrusters. This modification was done to obtain good results, since accelerometer measurements were not being used.

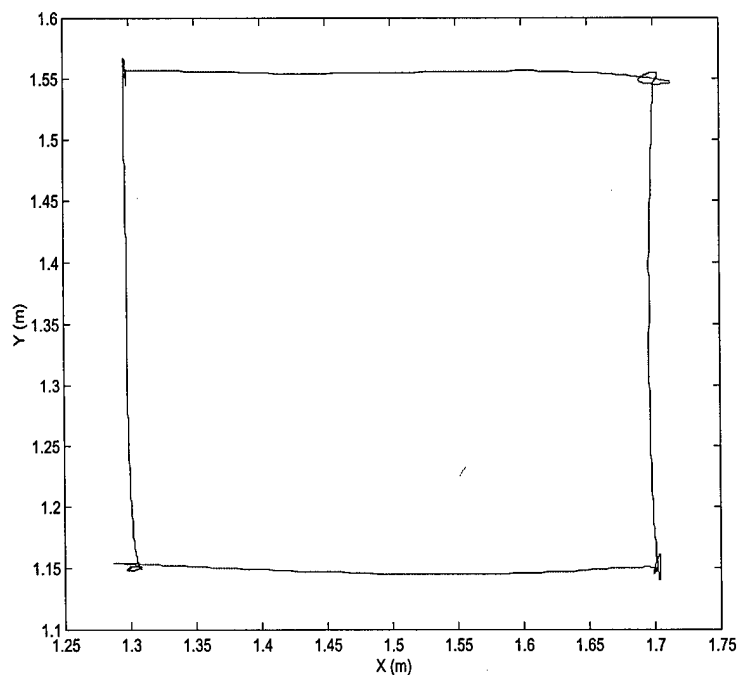### 5.2.1 Motion Observed:

Figure 5.1 Leader trajectory.

The leader's trajectory is plotted in Figure 5.1. We see that the leader traced out a clean square, with some oscillation at the corners. The oscillations are primarily due to inaccu-
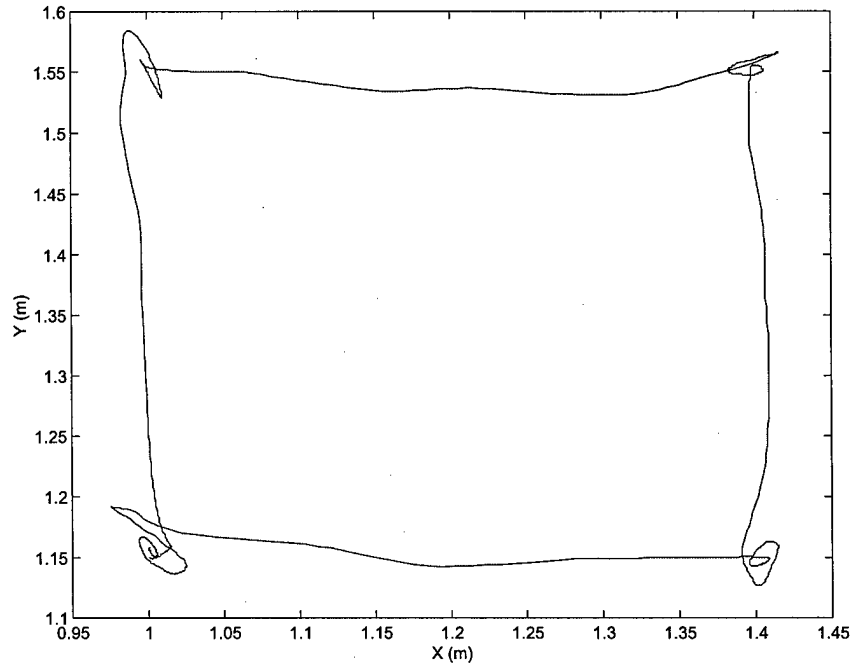
Figure 5.2 Follower trajectory.

racies in the state propagation that the SPHERE performs in between global metrology updates. Refinement of control parameters could also help to eliminate some of the overshoot. The follower's trajectory, in Figure 5.2, was significantly worse. The edges of the square are not straight, and the oscillations at the corners are larger. One might wonder why the follower's path is not identical to that of the leader, since it was controlling to the leader's state. The reason for this is that the leader was controlling to a target state that differs from its actual state. Therefore, the leader and follower were controlling to different target states. Any deviations from the desired trajectory were transmitted to the follower as its target state and were amplified in the follower. This is especially clear in Figure 5.3, which shows the positions of both the leader and follower along the x-axis with respect to time. We see that at the end of the plot, the leader overshot its final target state and oscillated before settling down. This overshoot and oscillation was magnified in the follower. A further reason for the poorer quality of the follower's trajectory is that the
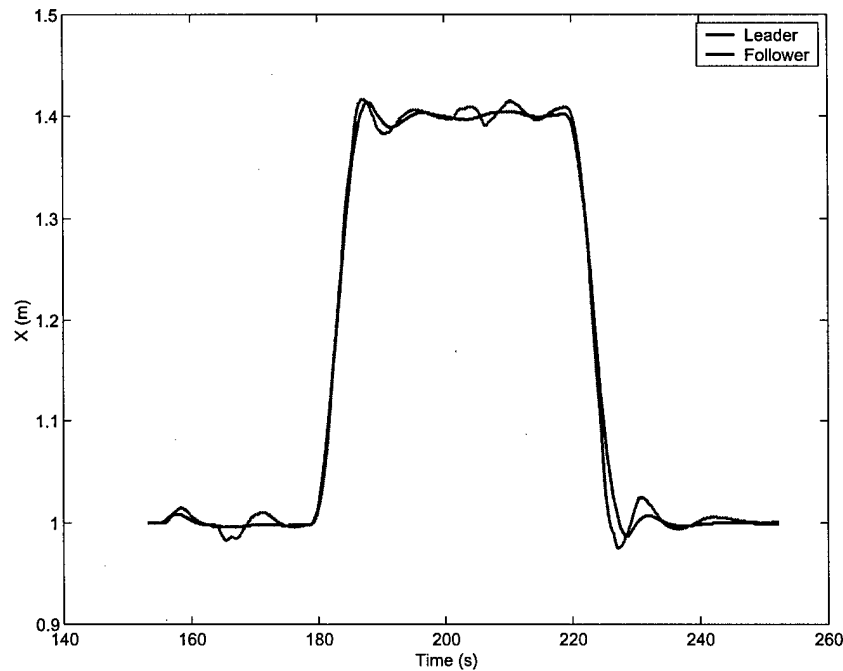
Figure 5.3   Time history of leader and follower position along x-axis (offset subtracted out).

leader's estimated state was not equal to its actual state. Recall the discussion in Section 4.4 of the fact that the SPHERE's position determination from global metrology measurements is never perfect, even when no noise is added to the measurements. Therefore, any errors in state estimation were also transmitted to the follower as its target state. This can be seen clearly at the start of the plot, where the leader made an initial adjustment because a slight error in its position estimate caused it to believe that it was not at the desired starting point. The follower, which had to deal with its own position estimation error, as well as that of the leader, had a harder time controlling to the desired state.

5.2.2 CPU Utilization

The CPU utilization characteristics for the leader and follower are shown in Figure 5.4, with the utilization broken down by process. These measurements were done using the
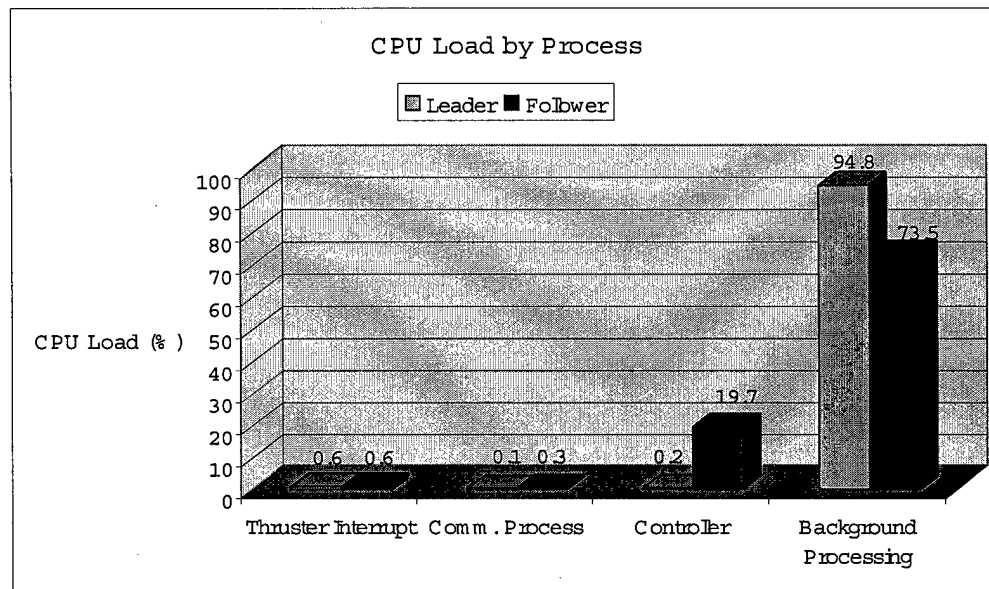
Figure 5.4 Leader and follower CPU loading comparison.

Profiler application that comes with OSE Illuminator, as described in Section 4.10. One notices that adding up the percentages for each SPHERE results in a total slightly less than 100%. This is due to the fact that there were other GRRDE and OSE processes executing that are not listed in the figure. We see that the thruster interrupt and communications process do not utilize much processing time. The interesting comparison is between the controller processes. While the leader's controller process required only 0.2% of the available processing time, the follower's utilized almost 20%, or 100 times more. This is explained by the fact that the follower received communications packets from the leader, while the leader did not. When extracting data from the arrays that hold communications data, there is a 4ms timeout. In other words, the follower keeps on extracting data from the array until such time that the array is continuously empty for 4ms. This is done because the communications process could be placing data into the array at the same time that the controller is extracting it. The timeout is meant to ensure that all of the data for a particular message will be received during the same pass through the controller code. The controller process was executing at a rate of 50Hz, corresponding to 20ms between consecutive acti-

vations. Therefore, the 4ms accounts for the 20% CPU load of the controller process. Any free time was used by background processing. Again, the majority of this processor utilization was due to communications timeouts.

5.2.3 Force History

For a separate run, where the same desired trajectory was used, the leader's force history was recorded in order to demonstrate the force and torque recording capabilities of the simulator. For this run, in order to obtain cleaner data, the leader was provided with perfect state information from the dynamics simulator. The force history is shown in
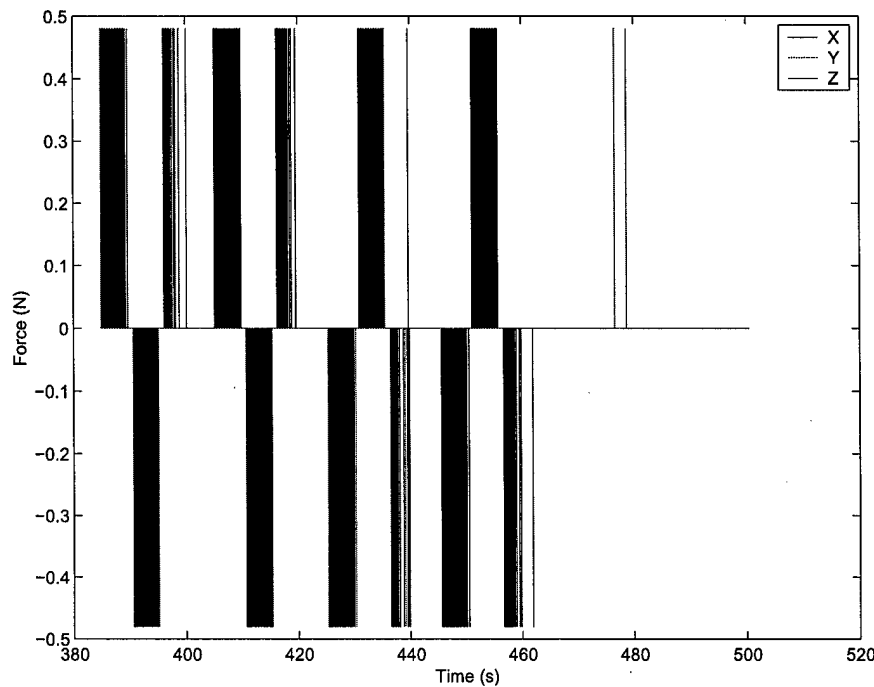
Figure 5.5 Force history for leader tracing out a square.

Figure 5.5. At the scale of this figure, most of the thruster firings appear on top of each other, giving the appearance of a long pulse, when they are all in fact very short pulses.

We can recognize the thruster firings associated with each of the sides of the square. The SPHERE started off by moving in the +Y direction, then in the +X direction, followed by -Y and -X. For each side, there was an acceleration period, followed by a deceleration, and some further thrusts that damped out any oscillations.

5.3 SPHERE-SPHERE Collision Simulation

A simple collision test was done with two SPHERES moving towards each other at an angle. No control was used for this test (ie. no thrusters were fired). Since their docking ports were not facing each other, they were expected to collide and bounce off of each other. Because the coefficient of restitution for SPHERE-SPHERE collisions was set at 0.5, the units were expected to have relative velocities of separation of half the magnitude of their relative velocity of approach. The motion observed during the test is shown in Figure 5.6. The SPHERES started at the bottom of the figure and moved upwards. They
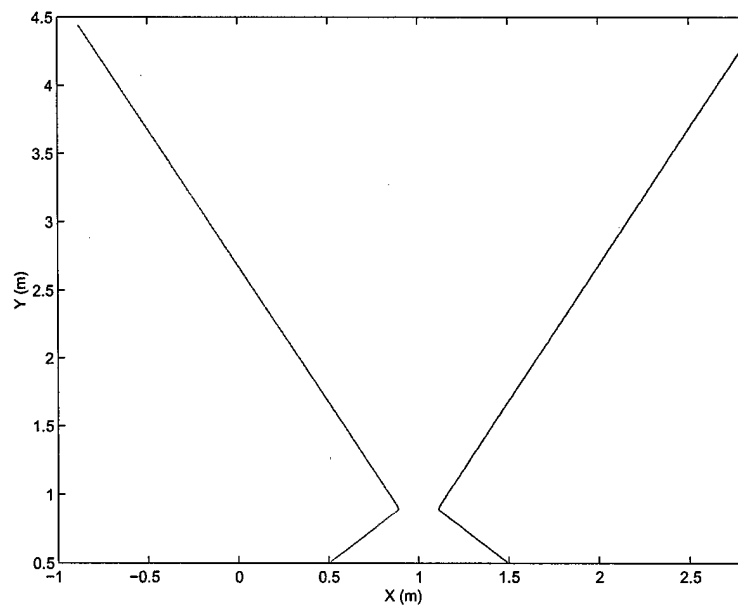
Figure 5.6 Motion for collision between two SPHERES.

collided when their centers were 20 cm apart, equal to two radii of the SPHERES. The units then fly apart with the expected change in their velocities.

5.4 Passive Docking Simulation

In order to test the docking capabilities of the GFLOPS SPHERES simulation, a simple test was conducted with two SPHERES moving towards each other with their docking ports facing. The SPHERES began with a large position offset along the X axis, as well as a 5 cm offset along the Y axis. They were given initial velocities parallel to the X axis (one of +5 cm/s, the other -50 cm/s). Again, no control was used for this test. Since they were offset along the Y axis, we expected the docked SPHERES to rotate about the Z axis due to conservation of angular momentum. In Figure 5.7, we see the X axis motion. The
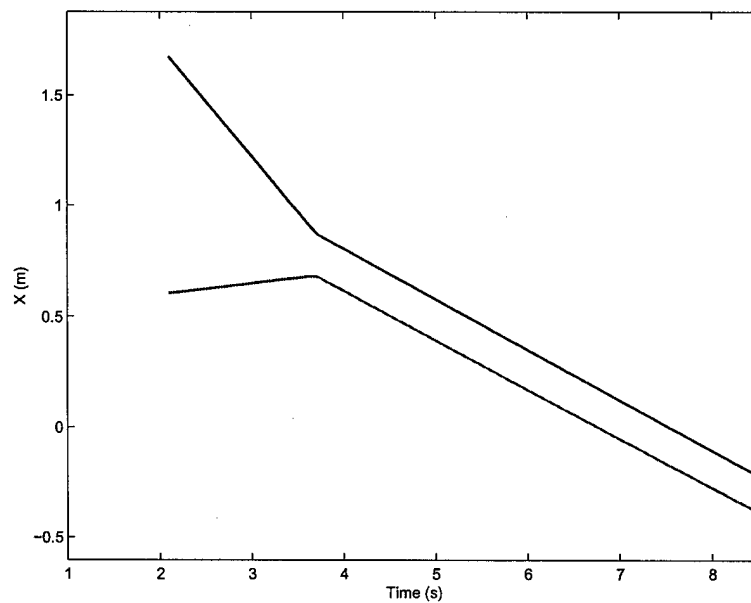
Figure 5.7 Motion along X axis for docking SPHERES.

SPHERE represented by the blue trace moved towards the other at a higher velocity. Once
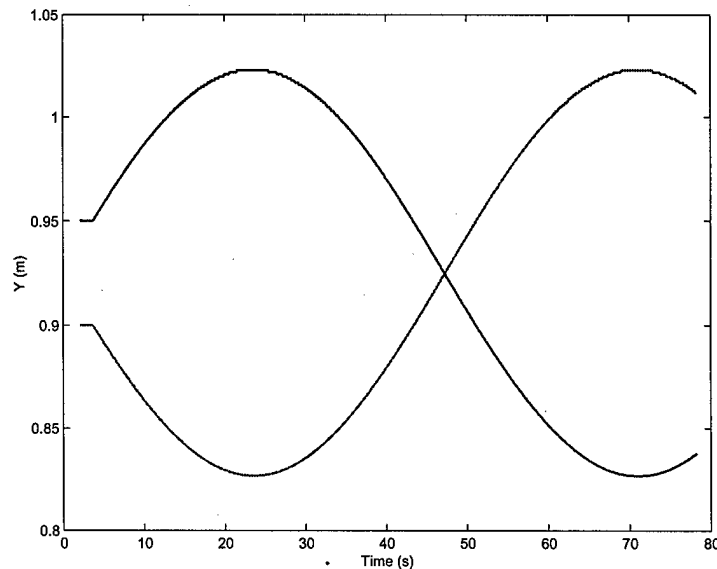
Figure 5.8 Motion along Y axis for docking SPHERES.

they collided, they stuck together with a distance of 20 cm between them. Conservation of momentum dictated that the faster moving SPHERE slowed down, while the other sped up by the same amount. Figure 5.8 depicts the Y axis motion of the SPHERES. After the collision, conservation of angular momentum resulted in rotation about the Z axis, which is reflected in the oscillating Y positions. The oscillations cannot be seen in Figure 5.7 because the time scale is much shorter than in Figure 5.8.

5.5 Cooperative Docking Simulation

A form of cooperative docking was tested with the GFLOPS SPHERES Simulator. It involved a leader satellite that was attempting to remain in one spot and was sending its state to a follower satellite. The follower, which started off with an initial offset in its orientation and in its position along the Y-axis, was supposed to dock with the leader. It was to do this by first rotating 90° about its Z-axis, then eliminating the difference in their

positions. Both the rotation and the translation parallel to the Y-axis were designed to take the form of a raised cosine. A raised cosine is a function of the form:

$$f(t) = f(t_0) + A\left(\frac{1 - \cos\omega(t - t_0)}{2}\right) \tag{5.1}$$

Figure 5.9 shows the shape that is expected when a raised cosine is used to reach a new
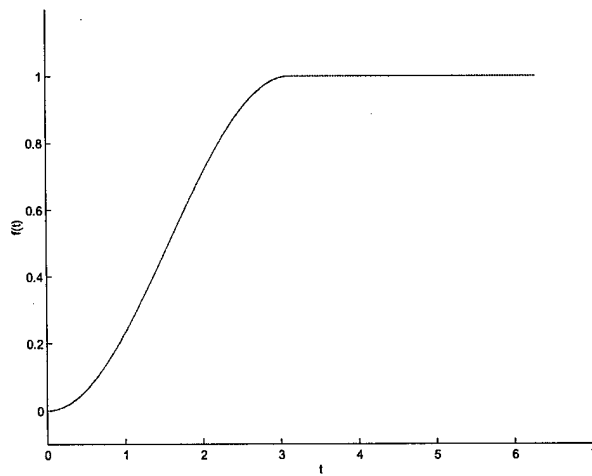
Figure 5.9   Raised cosine.

final value.

The same test had also been run on the MIT SSL air table in December 2000. The same code was used for the GSS test, to ensure that metrology processing and control of the SPHERE were handled identically. The motion of the follower for the air table and the GSS test is summarized in three figures. The SPHERE's internal state estimates were used as the data for these plots, because this is what was available for the air table test. For all previous plots in this chapter, truth data from the dynamics simulator was used.

Figure 5.10 shows the initial Z-axis rotation. Although the rotation on the air table had a
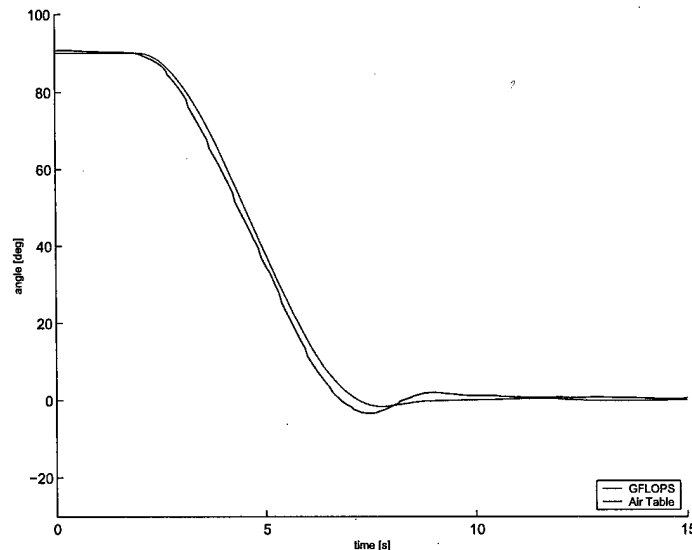
Figure 5.10   Comparison of Z-axis rotation for air table and GSS.

slightly larger overshoot, the curves are quite close to each other.

Figure 5.11 shows the motion parallel to the Y-axis as a function of time for both tests. Again, the curves are similar although, towards the end of the air table test, the SPHERE was unable to get a good global metrology measurement, which explains the horizontal line at the end of that test. The fact that the other SPHERE did not have this trouble indicates that the metrology simulator did not exclude enough global metrology measurements. This was essentially due to the fact that the maximum acceptable receiver angle was set to be too large. The discrete "stepped" appearance of the plots is due to the fact that the SPHERE's estimate of its position was only updated when it received a global metrology measurement. The fact that there are more steps in the GSS plot is further indication that the metrology simulator was providing too many good global metrology measurements.
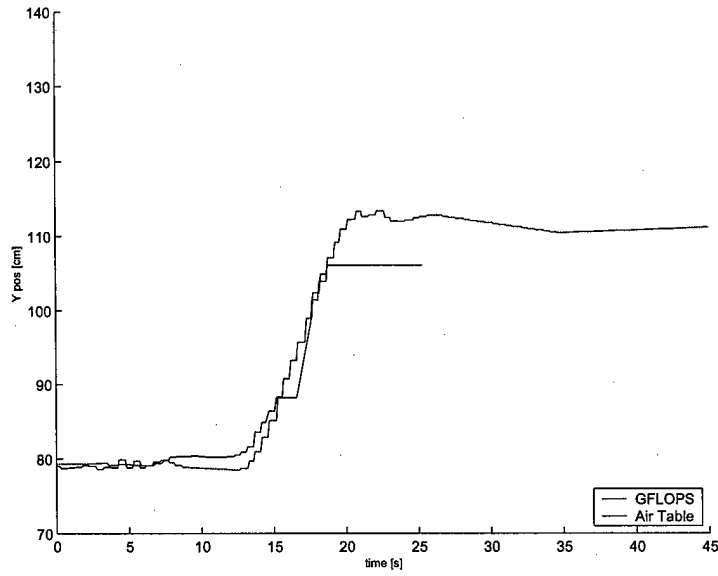
Figure 5.11 Comparison of motion parallel to Y-axis for air table and GSS.
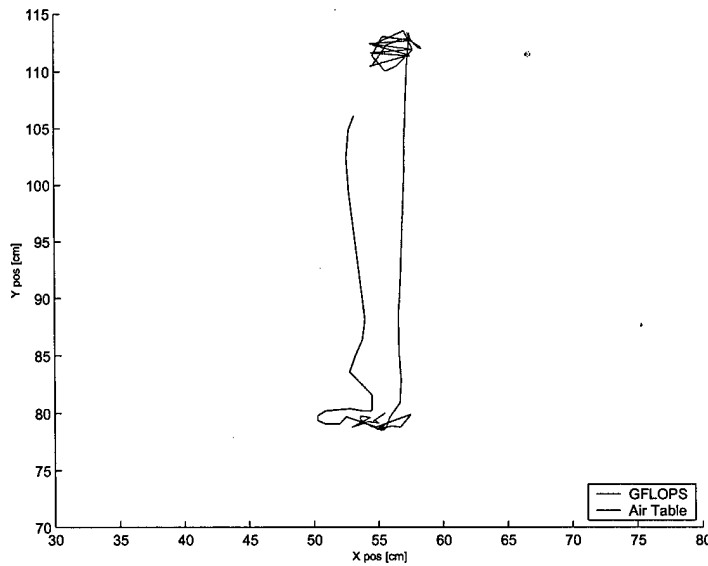
Figure 5.12 Comparison of XY plane motion for air table and GSS.

Figure 5.12 shows the motion parallel to the XY plane for both tests. The favorable global metrology measurements explain why the GFLOPS data is cleaner than the air table data. Their paths do show the same trends though. The oscillation at the end of GSS test is quite large because the follower was repeatedly bouncing off of the leader (their docking ports were not aligned).

5.6 Summary

This chapter presented some results of simulations that were run on the GFLOPS testbed. A leader-follower square profile simulation demonstrated several of the capabilities of the GSS, including inter-SPHERE communications, measurement of CPU utilization, and force recording. A simple collision between SPHERES and a passive docking simulation proved the ability of the GSS to detect and simulate collisions and docks. A direct comparison between cooperative docking motion obtained with the same control code on the air table and the GSS provided evidence of the usefulness of the simulation functions of the GSS.

Chapter 6

CONCLUSIONS

This thesis presented the GFLOPS SPHERES Simulator (GSS). This chapter begins by summarizing the main conclusions that were drawn in the previous chapters. It then analyzes the usefulness of the GSS with respect to the three control interfaces. Finally, it presents some suggestions for future work on the GSS.

6.1 Summary

6.1.1 SPHERES

The SPHERES testbed will allow for verification of satellite formation flight, autonomy and autonomous rendezvous and docking algorithms designed by external, "guest" scientists. SPHERES will operate in zero-gravity inside the International Space Station (ISS), an environment whose characteristics cannot be accurately reconstructed in the laboratory on Earth. There will be only 24 hours of flight time on ISS, which puts an onus on the SPHERES team to present the astronauts with flight code that will run correctly the first time.

There are six SPHERES subsystems: power, software, communications, metrology, avionics and propulsion. Of these six, software is the only one whose design and operation will vary for different guest scientist algorithms. It is also the only one that cannot be fully

tested on the laboratory air table, since some control algorithms cannot be tested with only three degrees of freedom.

6.1.2 GFLOPS

The GFLOPS testbed is an excellent simulation environment for SPHERES. It features 8 networked, embedded computers running the OSE real-time operating system. Loading separate programs onto different boards mimics the flight system, where separate programs are loaded onto different SPHERES. Because OSE was designed for distributed applications, implementing communications between SPHERES was relatively easy.

GFLOPS is a real-time testbed, so it is well suited to simulating a real-time system such as SPHERES. The SPHERES code runs with the same timing characteristics as in the flight system, so problems related to timing or synchronization can be discovered through simulation on the GSS. Also, the capability of measuring processor utilization can be very valuable for measuring and comparing the efficiency of guest scientist algorithms.

GFLOPS is further enhanced by the GFLOPS Rapid Real-Time Development Environment (GRRDE). Because GRRDE was designed to aid development of spacecraft flight code and simulation of distributed satellite systems, its features eased the development and testing of the GSS. Most notably, GRRDE helped in establishing communications links between software modules via contracts. Tools provided by GRRDE, such as atomic objects, were found to be useful, and the GRRDE module structure helped organize software modules.

6.1.3 GFLOPS SPHERES Simulator

The GSS solves the problem of testing SPHERES flight code in a zero-gravity, 6 DOF environment. By compiling flight code into SPHERE modules that can be run on the GFLOPS testbed, it allows testing with multiple SPHERES of guest scientist algorithms for formation flight, autonomy and docking. The GSS models thruster and metrology

characteristics, and propagates the dynamics of SPHERES units. It can handle collisions and docking between SPHERES, as well as user-applied disturbance forces.

Users can monitor the progress of simulations with a 3-D viewer and can receive telemetry or send commands to the SPHERES. Results of simulations can be saved and played back in the 3-D viewer at a later time. The OSE real-time operating system provides tools to analyze the processor utilization of SPHERES flight code.

6.1.4 Simulation Results

A leader-follower simulation was run, where the leader traced out a square profile and the follower attempted to execute the same profile by controlling to the leader's state (with some offset). This test demonstrated thruster, dynamics and metrology simulation as well as satellite-to-satellite (STS) communications. It further allowed us to test measurement of CPU utilization, yielding the interesting result that the follower's controller interrupt used much more processor time than the leader's, since it spent time waiting for state data to arrive from the master.

A collision simulation showcased the dynamics simulator's ability to handle SPHERE-SPHERE collisions. A passive docking simulation, where one SPHERE simply ran into the other, demonstrated the dynamics simulator's ability to detect and simulate docking, including conservation of linear and angular momentum. Results of tests of a cooperative docking algorithm on the MIT SSL air table and in the GFLOPS SPHERES Simulator were compared. The motion of the SPHERES was relatively similar, but the data indicated that measurement reception could be better modelled in the metrology simulator.

6.2 Suitability of the Simulator for the Control Interfaces

6.2.1 Standard Control Interface

The GFLOPS SPHERES Simulator is best suited to the Standard Control Interface. If a guest scientist places a list of maneuvers into the Standard Control Interface, they can be compiled and run with no modifications.

6.2.2 Direct Control Interface

The Direct Control Interface can be accomodated by the GSS. Since the controller process calls the SPHERES control code without modification, any changes to the control interrupt can be incorporated. Changes to the background processing can be accomodated, but would require some effort. Because background processing in the SPHERES flight code exists in the **main** function in the file main.c, along with various commands executed prior to the background processing loop that are not compatible with the GFLOPS testbed, the background processing code has to be manually inserted into the background process. However, this is generally not too difficult and consists of cutting and pasting code. It is not expected that guest scientists would make changes to the propulsion interrupt or the communications interrupt, since these perform lower-level functions that should be common to all guest scientist algorithms. Changes to these interrupts are possible, though they would have to be made manually in the SPHERES wrapper code.

The cooperative docking simulation (Section 5.5) provided a good test of the adaptability of the GSS. Because the control code for the algorithm was 1.5 years old, it did not follow the Standard Control Interface. The majority of global variables had different names or did not correspond to variables in the Standard Control Interface. Therefore, the cooperative docking simulation was an example of a use of the Direct Control Interface. Further complicating matters, metrology processing was handled entirely differently and differences in hardware configurations (eg. numbering of thrusters) were present.

The time spent modifying the SPHERE module wrapper code to accomodate the cooperative docking simulation was logged. A total of 4.5 hours was spent getting the SPHERE module to correctly compile. For someone with less familiarity with the GSS, the time required would have been much longer. The differences in metrology also required some changes to the metrology simulator, which are not included in the 4.5 hours. Furthermore, several days were spent debugging problems that arose.

The cooperative docking simulation is an extreme example of the use of the Direct Control Interface, due to the magnitude of the deviation from the Standard Control Interface. Nonetheless, it illustrates the fact that testing code that does not conform to the Standard Control Interface can pose some serious challenges.

6.2.3 Custom Control Interface

The GSS is not well suited to the Custom Control Interface. This interface has essentially no rules associated with it, so it is not difficult to see that the GSS, running on different hardware with a different operating system, cannot have the flexibility to easily incorporate Custom Control Interface code. That is not to say that custom code cannot be simulated on the GFLOPS testbed. But it would require considerable effort to change such things as the number, type, and purpose of the processes that exist in the SPHERE module, and to debug problems that arise. To do so would require in-depth knowledge of OSE, GRRDE, and the GSS.

6.3 Future Work

All of the main elements of the GFLOPS SPHERES Simulator could be improved to some extent. There are features missing that would improve the accuracy or usability of the simulator. These areas for future work will now be discussed.

6.3.1 Dynamics Simulator

One inaccuracy in the dynamics simulator is that it does not take into account the decrease in total mass of a SPHERE due to decreasing propellant. Propellant accounts for approximately 5% of a SPHERE's total mass [Miller, 2002]. The amount of propellant used could be kept track of by the thruster simulator, since it knows when thrusters are turned on or off. This information could be sent to the dynamics simulator, which would incorporate this effect into the state propagation.

The dynamics simulator has some deficiencies related to collisions and docking. These both work fine up until the point that two SPHERES dock. After that point, if there is a third SPHERE present in the simulation, the simulator is not capable of handling a dock or collision between this third SPHERE and one of the others that are already docked. Nonetheless, this could easily be implemented.

6.3.2 Metrology Simulator

The detail to which the metrology simulator represents global metrology signals could be improved. In particular, the metrology simulator currently sends distances for all beacon-receiver pairs for which the transmitter and receiver angles are both less than 90°. However, hardware testing has shown that measurements are often not received for angles greater than 60°. If enough detailed hardware calibration were done, we could characterize the dependence of measurement reception on transmitter angle, receiver angle, and distance, and build a probabilistic model. Doing this with some preexisting data was explored, but there was not enough variety in distances and angles to yield a useful calibration. Data was only available with for a SPHERE in the middle of the laboratory test space, at orientations that differed by 90°. Since the locations of the receivers are symetric for 90° rotations, the new orientations yielded no new data.

An issue that is not addressed in the metrology simulator is that of loss of global metrology measurements due to body blockage. Body blockage occurs when one SPHERE is

directly in the path between a beacon and one of the other SPHERE's receivers. In such a case, that measurement should not be sent, since it would not be received in practice. This capability would not be difficult to implement. Since the positions of all SPHERES are known, the perpendicular distance between each SPHERE and the line connecting a beacon to a receiver could be easily calculated. Then, we could determine if this distance is less than the radius of a SPHERE. If so, blockage would occur for that beacon-receiver pair.

A further area in which the operation of the metrology simulator could be improved is in the way that it returns measurements. In the actual SPHERES flight code, the CPU receives IMU and global metrology measurements in the same way as STS or STL communications. The data arrives as individual bytes in the communications interrupt. This is not the way that it happens in the simulation. Here, each complete IMU or global metrology measurement is sent in an OSE signal to the SPHERE background process. Changing this to be compatible with the format expected by unmodified SPHERES flight code would require some effort, but could clearly be done.

6.3.3 Thruster Simulator

For the thruster simulator, better models of the thrusters could be achieved. Currently, the nominal thrust level for each thruster on each SPHERE is assumed to be the same. This is not accurate, because slight differences in machining the thruster nozzles result in differences in the thrust produced from each nozzle. It could be valuable to be able to set the thrust level for each thruster independently. Another issue that has not been addressed is encountered when two SPHERES dock together. Some of the thrusters on the docking panels will be directly facing or touching the other SPHERE. How does this affect the net thrust experienced by the docked SPHERES?

6.3.4 Communications Manager

For both STS and STL communications, there is a maximum bit rate of 19.2 kbps that is set by the communications hardware [Otero, 2000]. For STL communications, the GSS is not able to support this rate. However, STS communications on the GFLOPS testbed can occur at 100 Mbps. To make STS communications more representative of the actual hardware, it could be possible to limit the communications bandwidth. This could consist of keeping a moving average of the rate being achieved by the currently transmitting SPHERE. If sending the next byte would make the average communications rate higher than the maximum, then we would wait before sending this byte. To avoid changes to the SPHERES flight code, this new functionality should reside in a different module. The most obvious place to put it would be in the communications manager. That would mean routing STS communications, in addition to STL communications, through this module, which should not be a problem.

6.3.5 3-D Viewer

The 3D viewer is well suited for playback of short simulations. However, for longer simulations, of 10 minutes duration for example, the viewer is missing some features that would be beneficial. Often, there is one short fraction of the simulation that is of greater interest than the rest. An example is docking, where the few seconds leading up to the dock might be the most interesting. If one wants to analyze these few seconds in detail, from different angles and zoom factors, it is clearly not convenient to have to replay the entire 10 minutes of the simuation just to see these important few seconds multiple times. One would expect to have a "pause" button and a "play" button that allow the user to stop the playback, view the scene form different vantage points, then start it again when desired. In addition, a "slider" control that allows one to move to any point in the simulation by moving the slider forwards or backward would be very useful. Another possible improvement includes drawing the global frame axes to help the user visualize the test space orientation. Furthermore, it could be useful to have an optional "trace" function that leaves small dots along the path traced out by the SPHERE, to allow for visualization of

the path. Finally, adding some identification marks on the panels of the SPHERES, in order to discern their orientations, would be helpful. This would be especially true for docking, where one wants to be able to identify their docking ports. All of these suggestions could be implemented fairly easily.

6.3.6 CPU Utilization

As mentioned in Section 4.10, we cannot draw any conclusions about the absolute processor utilization on the flight hardware DSP unless some type of calibration is performed with the same code on the GFLOPS processors and the DSP. Even then, much care would have to be taken in attempting to makes conclusions about DSP processor utilization. However, the insight gained in making the calibration could be quite valuable. For example, consider the case of an algorithm for formation flying, docking, or some type of autonomy, that can only be tested in a 6 DOF environment. We could not test it on the MIT SSL air table and might not be able to tell if the algorithm can run safely within the constraints of the DSP's performance. This might also be the case for an algorithm that we can test in 3 DOF, but which breaks down into much simpler calculations in this environment.

Benchmarking has been performed before with the GFLOPS testbed to determine the relative running time of various floating point computations. The technique commonly used is to perform the same calculation a large number of times (say 100000), and then compute the average time for the calculation. This technique could be easily extended to measure the running time of a longer algorithm, such as the routine used to determine the SPHERE'S state from global metrology measurements. Performing the same benchmark on the GFLOPS and SPHERES hardware would yield some insight into the relationship between processor utilization on the two systems.

REFERENCES

[AFRL, 2002] "TechSat 21: Next Generation Space Capabilities", AFRL website, `http://www.vs.afrl.af.mil/techprogs/techsat21/ngsc.html`, February 2002.

[DARPA, 2002] "Orbital Express Space Operations Architecture / ASTRO", DARPA website, `http://www.darpa.mil/tto/programs/astro.html`, February 2002.

[ENEA, 1998] *OSE Documentation, Volumes 1-4*, Enea OSE Systems AB, Sweden, 1998.

[Enright, 2002] Enright, J. P., *A Flight Software Development and Simulation Framework for Advanced Space Systems*, MIT Ph.D. Thesis in Aeronautics and Astronautics, June 2002.

[Fish, 1994] Fish, P. J., *Electronic Noise and Low Noise Design*, McGraw-Hill, New York, 1994.

[HilstadA, 2002] Hilstad, M. O., *The SPHERES Guest Scientist Program Interface (version 1.0)*, MIT Space Systems Laboratory document, May 2002.

[HilstadB, 2002] Hilstad, M. O., *Validation of Distributed Spacecraft Control Algorithms and Topologies using a Multi-Vehicle Formation Flying Testbed*, MIT S.M. Thesis in Aeronautics and Astronautics, June 2002.

[Hughes, 1986] Hughes, P. C., *Spacecraft Attitude Dynamics*, John Wiley and Sons, New York, 1986.

[IBM, 2001] *PowerPC 740 and PowerPC 750 Microprocessor Datasheet, Version 1.1*, IBM Microelectronics Division, October 2001.

[JPLA, 2002] "Starlight", JPL website, `http://starlight.jpl.nasa.gov`, February 2002.

[JPLB, 2002] "Terrestrial Planet Finder", JPL website, `http://tpf.jpl.nasa.gov`, February 2002.

[JSC, 2002] "KC-135 Reduced Gravity Research", Johnson Space Center website, `http://jsc-aircraft-ops.jsc.nasa.gov/kc135/index.html`, February 2002.

[Meriam, 2002] Meriam, J. L. and Kraige, L. G., *Engineering Mechanics: Dynamics*, 5th ed., John Wiley and Sons, 2002.

[Miller, 2002] "SPHERES Critical Design Review", Powerpoint presentation on the MIT SSL website, `http://ssl.mit.edu`, February 2002.

[Otero, 2000] Otero, A. S., *The SPHERES Satellite Formation Flight Testbed: Design and Initial Control,* MIT S.M. Thesis in Aeronautics and Astronautics, August 2000.

[TI, 2000] *TMS320C6701 Floating-Point Digital Signal Processor,* Texas Instruments Literature Number SPRS067RE, May 2000.

Appendix A

GFLOPS SPHERES SIMULATOR SOURCE CODE

A.1 Dynamics Simulator

A.1.1 Sph_propagator.h

```
#ifndef __SPH_PROPAGATOR__
#define __SPH_PROPAGATOR__

#include <siglib.h>
#include "sph_int_object.h"
#include "Spheres_constants.h"

class CPropagator:public CIntObject {
public:
    CPropagator();
    void Rezero();
    void SetDisturbance(double dForce[3], double dTorque[3], int iDuration);
    bool IsDisturbanceOn();
    void ZeroDisturbance();
    void ZeroTorques();
    void ZeroForces();

    void SetInvMass(double dInvMass);
    void SetInertia(double dI[3][3]);
    void GetInertia(double dI[3][3]);
    void GetInvInertia(double dI[3][3]);
    void SetPrincipalInertia(double * dI);
    bool Inertia_Is_Set();

    void SetGCState(double dState[STATE_LENGTH]);
    void SetCMState(double dState[STATE_LENGTH]);
    void GetState(double dState[STATE_LENGTH]);
    void GetCMExtendedState(double dState[EXTENDED_STATE_LENGTH]);
    void GetExtendedState(double dState[EXTENDED_STATE_LENGTH]);
    void GetPosition(double dPos[3]);
    void GetPosVel(double dPos[3], double dVel[3]);

    void InterpToTime(double t);
    void InterpToTime(struct TimePair *tp);
    void SetTorqueThrust(double dTorque[3], double dThrust[3], double dTime);
    void GetForceFromThrusters(double dForce[3]);
    int IntegrateToTime(double dTime);

    //Debug Methods
    int GetNumLoops();

protected:
    //eta, epsilon(0,1,2), omega(0,1,2), pos(0,1,2), vel(0,1,2)
    double dX1[STATE_LENGTH];
    double dX2[STATE_LENGTH];
```

```
    double dX_out[STATE_LENGTH];

        //Constant pointers to first elements of the various components of the state
    double *const m_pdQuat;
    double *const m_pdRate;
    double *const m_pdPos;
    double *const m_pdVel;

    double dWorstTol;
    double m_dws[14][STATE_LENGTH];
    double m_dC[24];

    double m_dI[3][3];
    double m_dInvI[3][3];
    double m_dInvMass;

        //Position of the geometric center relative to the center of mass (in body frame coordinates)
    double m_dGCpos[3];

    //3 body frame torques (x, y, z)
    double m_dBodyTorque[3];
    //3 body frame translational forces (x, y, z)
    double m_dBodyForce[3];
    //3 inertial frame translational forces (x, y, z)
    double m_dInertialForce[3];
    //Disturbance force commanded from 3-D viewer application
    double m_dDistForce[3];
    //Disturbance torque commanded from 3-D viewer application
    double m_dDistTorque[3];
    double m_dDisturbStartTime;
    double m_dDisturbDuration;
    bool m_bDisturbanceOn;

    void normQ(double* dX);

    int CombinedDerivFunc(int *n_eqn, double *t,double *X, double * X_prime);
    int Integrate(bool bFromNow=false);
    void ConvertThrustToInertialFrame();
    void CalcInvI();

    //Debug Info
    int iNumLoops;

private:
};

#endif
```

A.1.2 Sph_propagator.cpp

```
#include "sph_propagator.h"

#include <math.h>
#include <string.h>

#ifndef __NEED_TP_DEFN__
#define __NEED_TP_DEFN__
#endif
#include "gflp_obt_conv.h"
#include "quick_vectors.h"

////////////CPropagator////////////////////////////////

/*
 * Constructor
 */
CPropagator::CPropagator()
    //Initializer list to initialize constant pointers
:m_pdQuat(&dX_out[SIM_QUAT_1]), m_pdRate(&dX_out[SIM_RATE_X]), m_pdPos(&dX_out[SIM_POS_X]),
                m_pdVel(&dX_out[SIM_VEL_X]) {

    int i;
    m_dInvMass = INV_MASS;

    //Make posvel all zero
```

```
    //Need a valid initial quaternion: eta=1, epsilon=0;
    for (i=0;i<STATE_LENGTH;i++)
    {
        dX1[i]=0.0;
        dX2[i]=0.0;
        dX_out[i]=0.0;
    }
    dX1[SIM_QUAT_1]=1.;
    dX2[SIM_QUAT_1]=1.;
    dX_out[SIM_QUAT_1]=1.;

    //Number of state elements
    iN=STATE_LENGTH;
    iNw=STATE_LENGTH;

    //Initialize the inertia matrix
    int j;
    for (i=0;i<3;i++)
    {
        for (j=0;j<3;j++)
        {
            m_dI[i][j]=0.0;
            m_dInvI[i][j]=0.0;
        }
    }
    m_dI[0][0]=INERTIA_XX;
    m_dI[1][1]=INERTIA_YY;
    m_dI[2][2]=INERTIA_ZZ;
    CalcInvI();

    for (i=0; i<3; i++) {
        m_dBodyTorque[i]=0.;
        m_dBodyForce[i]=0.;
        m_dInertialForce[i]=0.;
    }
    ZeroDisturbance();
    m_dDisturbStartTime = 0.0;
    m_dDisturbDuration = 0.0;

    //Set position of geometric center w.r.t center of mass in body frame coordinates
    m_dGCpos[0] = -CM_POS_X;
    m_dGCpos[1] = -CM_POS_Y;
    m_dGCpos[2] = -CM_POS_Z;

    //Set things up for integrator
    dWorstTol=0.;
    X_dot=(DerivFunc) &CPropagator::CombinedDerivFunc;
}


/*
 * Set 3x3 inertia matrix of satellite
 */
void CPropagator::SetInertia(double dI[3][3])
{
    memcpy(m_dI,dI,sizeof(double)*9);
    CalcInvI();
}


/*
 * Calculate the inverse of the 3x3 inertia matrix
 * WARNING: ASSUMES I is diagonal
 */
void CPropagator::CalcInvI()
{
    SFLOAT sfInvI[3][3], sfTempI[3][3], sfIndex[3][3], sfScaling[3][3];
    SFIX sfRow[3][3];
    siglib_numerix_SMXInverse((SFLOAT*)m_dI, (SFLOAT*)sfInvI, (SFLOAT*)sfTempI, (SFLOAT*)sfIndex,
                  (SFIX*)sfRow, (SFLOAT*)sfScaling, 3);
    int i, j;
    for(i=0; i<3; i++) {
        for (j=0; j<3; j++) {
            m_dInvI[i][j] = sfInvI[i][j];
        }
    }
}
```

```
/*
 * Get copy of 3x3 inertia matrix of satellite
 */
void CPropagator::GetInertia(double dI[3][3])
{
    memcpy(dI,m_dI,sizeof(m_dI));
}


/*
 * Set full state of satellite
 * CAREFUL: This sets the center of mass state directly.
 *          It does not take into account the difference in
 *          position between CM and geometric center.
 */
void CPropagator::SetCMState(double dState[STATE_LENGTH])
{
    normQ(&dState[SIM_QUAT_1]);
    memcpy(dX2,dState,sizeof(dX2));
    memcpy(dX_out,dState,sizeof(dX_out));
}


/*
 * Set full state of satellite by specifying coordinates of the geometric center.
 */
void CPropagator::SetGCState(double dState[STATE_LENGTH]) {
    SetCMState(dState);
        //Now overwrite the posvel info to take into account CM offset
    double dTemp[3];
    quat_rotate_out(m_pdQuat, m_dGCpos, dTemp);
    dX2[SIM_POS_X] = dX_out[SIM_POS_X] = dState[SIM_POS_X] - dTemp[0];
    dX2[SIM_POS_Y] = dX_out[SIM_POS_Y] = dState[SIM_POS_Y] - dTemp[1];
    dX2[SIM_POS_Z] = dX_out[SIM_POS_Z] = dState[SIM_POS_Z] - dTemp[2];

        //Take into account that the body frame is a rotating reference frame.
    double wxr[3];
    crossProduct(m_pdRate, m_dGCpos, wxr);
    quat_rotate_out(m_pdQuat, wxr, dTemp);
    dX2[SIM_VEL_X] = dX_out[SIM_VEL_X] = dState[SIM_VEL_X] - dTemp[0];
    dX2[SIM_VEL_Y] = dX_out[SIM_VEL_Y] = dState[SIM_VEL_Y] - dTemp[1];
    dX2[SIM_VEL_Z] = dX_out[SIM_VEL_Z] = dState[SIM_VEL_Z] - dTemp[2];
}


/*
 * Interpolate the state vector to time t, starting from time dT1
 * RESULT:dT_out set to t
 *           dX_out interpolated up to time t
 *
 */
void CPropagator::InterpToTime(double t)
{
    dT_out=t;
    double dTemp_t_int=dT2-dT1;
    intrp_(&iN,&dT1,dX1,&dT_out,dX_out,&dTemp_t_int,&iNw,m_dws[0]);
    normQ(&dX_out[SIM_QUAT_1]);
}

void CPropagator::InterpToTime(struct TimePair * tp)
{
    InterpToTime(tp2dbl(tp));
}


/*
 * Propagates the state vector from time dT2 to dTime
 * Will not integrate if dTime < dT2
 */
int CPropagator::IntegrateToTime(double dTime) {
    if (dTime > dT2) {
        dDelta_t = dTime - dT2;
        Integrate(false);
        return 0;
    }
    else return 1;
}
```

```
/*
 * Propagates the state vector from time dT1 to dT2 = dT1 + dDelta_t
 * PARAMETERS:bFromNow= true(reintegrate for rest of same timestep)
 *                    = false(integrate next timestep)
 */
int CPropagator::Integrate(bool bFromNow)
{
        //Make sure thrust takes into account current orientation of sphere
    //iInd=1;
    double dOldCurTime;//time from which we are integrating from
    double dCurTol=dTol;//current error tolerance that integrator is trying to maintain

    if (bFromNow)
    {    //Reintegrate for rest of same timestep
        dT1=dT_out;
        memcpy(dX1,dX_out,sizeof(dX1));
        memcpy(dX2,dX_out,sizeof(dX2));
        //Account for weird convergence behaviour
        iInd=1;
        memset(m_dws[0],0,sizeof(m_dws));
        memset(m_dC,0,sizeof(m_dC));
        //m_dC[6]=MAX_INT_FCN_EVALS;
        m_dC[2]=0.;//HMIN_ORBIT;
        dOldCurTime=dT_out;

    }
    else
    {
        //Integrate next step
        dT1=dT2;
        dT2=dT1+dDelta_t;
        memcpy(dX1,dX2,sizeof(dX1));
        //Account for weird convergence behaviour
        iInd=1;
        memset(m_dws[0],0,sizeof(m_dws));
        memset(m_dC,0,sizeof(m_dC));
        //m_dC[6]=MAX_INT_FCN_EVALS;
        m_dC[2]=0.;//HMIN_ORBIT;
        dOldCurTime=dT1;
        dT_out=dT2;
    }
    if (m_bDisturbanceOn && dT1 >= m_dDisturbStartTime + m_dDisturbDuration) ZeroDisturbance();

    //move X2 (of prev step) to X1
    //memcpy(dX1,dX2,sizeof(dX1));
    //integrate
    bool bWorking=true;
    iNumLoops = 0;
    while (bWorking)
    {
        dverk_(&iN, X_dot,&dT1,dX2,&dT2,&dCurTol,&iInd,m_dC,&iNw,m_dws[0]);
        iNumLoops++;

        if(iInd==RK_ERROR_ERRREQ)
        {
            //Retry with higher TOL. Rather ad hoc
            dT1=dOldCurTime;
            dCurTol *= 10.;
            dWorstTol=dCurTol;
            memcpy(dX2,dX1,sizeof(dX1));
            iInd=1;
            memset(m_dws[0],0,sizeof(m_dws));
            //memset(m_dC,0,sizeof(m_dC));
            //m_dC[6]=MAX_INT_FCN_EVALS;
            //m_dC[2]=0.;
            memset(m_dws[0],0,sizeof(m_dws));
        }
        else
        {
            bWorking=false;
        }

    }
    //dT1=dT2-dDelta_t;
    dT1=dOldCurTime;
```

```
    if (bFromNow)
    {
//    InterpToTime(dOldCurTime);
    }
    else
    {
        //dX2 is copied instead of dX1 to cover the case that intrp_ is never used
        memcpy(dX_out,dX2,sizeof(dX2));
    }

    normQ(&dX_out[SIM_QUAT_1]);//added 10/22/2002
    return iInd;
}


/*
 * Used to find the derivatives of the 13 state variables.
 * PARAMETERS:X= current state vector
 * RESULT:X_prime= set to derivative of X
 */
int CPropagator::CombinedDerivFunc(int *n_eqn, double *t,double *X, double * X_prime) {
    double dDistOn = 0.0;
    //Must use >= and <= instead of > and < or will crash
    if (m_bDisturbanceOn) {
        dDistOn = 1.0;
    }
    //ROC for quaternion (notation for euler parameters)
    //eta_dot
    X_prime[SIM_QUAT_1]=-
                .5*(X[SIM_QUAT_2]*X[SIM_RATE_X]+X[SIM_QUAT_3]*X[SIM_RATE_Y]+X[SIM_QUAT_4]*X[SIM_RATE_
                Z]);
    //epsilon_dot
    X_prime[SIM_QUAT_2]=.5*(X[SIM_QUAT_1]*X[SIM_RATE_X]-
                X[SIM_QUAT_4]*X[SIM_RATE_Y]+X[SIM_QUAT_3]*X[SIM_RATE_Z]);
    X_prime[SIM_QUAT_3]=.5*(X[SIM_QUAT_4]*X[SIM_RATE_X]+X[SIM_QUAT_1]*X[SIM_RATE_Y]-
                X[SIM_QUAT_2]*X[SIM_RATE_Z]);
    X_prime[SIM_QUAT_4]=.5*(-
                X[SIM_QUAT_3]*X[SIM_RATE_X]+X[SIM_QUAT_2]*X[SIM_RATE_Y]+X[SIM_QUAT_1]*X[SIM_RATE_Z]);

    //Rate of change of angular velocity
    double dTemp[3], dTemp2[3];
    siglib_numerix_SMXMultiply((SFLOAT*)m_dI, (SFLOAT*)(&X[SIM_RATE_X]), (SFLOAT*)dTemp, 3, 3, 1);
    crossProduct(&X[SIM_RATE_X], dTemp, dTemp2);
    int i;
    for (i=0; i<3; i++) {
        dTemp[i] = m_dBodyTorque[TORQUE_X+i] + dDistOn*m_dDistTorque[i] - dTemp2[i];
    }
    siglib_numerix_SMXMultiply((SFLOAT*)m_dInvI, (SFLOAT*)dTemp, (SFLOAT*)(&X_prime[SIM_RATE_X]), 3, 3,
                1);

    //Rate of change of position
    X_prime[SIM_POS_X]=X[SIM_VEL_X];
    X_prime[SIM_POS_Y]=X[SIM_VEL_Y];
    X_prime[SIM_POS_Z]=X[SIM_VEL_Z];

    //Rate of change of velocity
    quat_rotate_out(&X[SIM_QUAT_1],m_dBodyForce, m_dInertialForce);
    X_prime[SIM_VEL_X]=m_dInvMass*(m_dInertialForce[THRUST_X] + dDistOn*m_dDistForce[0]);
    X_prime[SIM_VEL_Y]=m_dInvMass*(m_dInertialForce[THRUST_Y] + dDistOn*m_dDistForce[1]);
    X_prime[SIM_VEL_Z]=m_dInvMass*(m_dInertialForce[THRUST_Z] + dDistOn*m_dDistForce[2]);

    return 0;
}

/*
 * Normalize the quaternion part of a state vector.
 * PARAMETERS:X= state vector
 */
void CPropagator::normQ(double* dX)
{
    double dNorm=1./sqrt(dX[0]*dX[0]+dX[1]*dX[1]+dX[2]*dX[2]+dX[3]*dX[3]);
    if (dX[0] < 0.0) {
        dX[0] *= -dNorm;
        dX[1] *= -dNorm;
        dX[2] *= -dNorm;
```

```
            dX[3] *= -dNorm;
    }
    else {
            dX[0] *= dNorm;
            dX[1] *= dNorm;
            dX[2] *= dNorm;
            dX[3] *= dNorm;
    }
}


/*
 * Check if the inertia has been set.
 * RETURNS:true if the Ixx component of the inertia matrix > 0.
 */
bool CPropagator::Inertia_Is_Set()
{
    //return (vect_mag3(m_dI)>.0);
    return m_dI[0][0]>.0;
}


/*
 * Set principal inertia values
 * PARAMETERS:dI= 3 element array containing principal inertia values
 */
void CPropagator::SetPrincipalInertia(double dI[3])
{
    m_dI[0][0]=dI[0];
    m_dI[1][1]=dI[1];
    m_dI[2][2]=dI[2];
}


/*
 * 3 body frame angular forces and 3 body frame translational forces
 * dTime: time at which these forces and torques are applied
 */
void CPropagator::SetTorqueThrust(double dTorque[3], double dThrust[3], double dTime)
{
        //First propagate the state up to now
    IntegrateToTime(dTime);
        //Now save the torque/thrust values so they can be taken into account
        //for the next integration
    for (int i=0; i<3; i++) {
        m_dBodyTorque[i] = dTorque[i];
        m_dBodyForce[i]  = dThrust[i];
    }
#ifdef ONE_G
    m_dBodyTorque[TORQUE_X] = 0.;
    m_dBodyTorque[TORQUE_Y] = 0.;
    m_dBodyForce[THRUST_Z] = 0.;
#endif
}


/*
 * Updates the m_dInertialForce array to take into account the new body orientation
 * Since the thrust set in SetTorqueThrust(...) is a body frame thrust but the SPHERE
 * might be rotating.
 */
void CPropagator::ConvertThrustToInertialFrame() {
    quat_rotate_out(m_pdQuat,m_dBodyForce, m_dInertialForce);
}



/*
 * Returns the position of the geometric center of sphere.
 * Rgc = Rcm + R(cm -> gc)
 * whereRgc= position of geometric center
 *           Rcm =position of center of mass
 *           R(cm -> gc)= vector from cm to gc
 */
void CPropagator::GetPosition(double dPos[3])
{
        //Rotate R(cm -> gc) from body frame coordinates into inertial coordinates
    quat_rotate_out(m_pdQuat, m_dGCpos, dPos);
    dPos[0] += dX_out[SIM_POS_X];
    dPos[1] += dX_out[SIM_POS_Y];
    dPos[2] += dX_out[SIM_POS_Z];
```